Combination of Abstract Domains for Logic Programming: Open Product and Generic Pattern Construction

Agostino Cortesi
Ca' Foscari University
via Torino 155
I-30170 Venezia (Italy)
`cortesi@dsi.unive.it`

Baudouin Le Charlier
University of Namur
21 rue Grandgagnage
B-5000 Namur (Belgium)
`ble@info.fundp.ac.be`

Pascal Van Hentenryck
Univ. Catholique de Louvain
2, Place Sainte-Barbe
B-1348, Louvain-la-Neuve (Belgium)
`pvh@info.ucl.ac.be`

1 Abstract

Abstract interpretation is a systematic methodology to design static program analysis which has been studied extensively in the logic programming community, because of the potential for optimizations in logic programming compilers and the sophistication of the analyses which require conceptual support. With the emergence of efficient generic abstract interpretation algorithms for logic programming, the main burden in building an analysis is the abstract domain which gives a safe approximation of the concrete domain of computation. However, accurate abstract domains for logic programming are often complex not only because of the relational nature of logic programming languages and of their typical interprocedural control-flow, but also because of the variety of analyses to perform, their interdependence, and the need to maintain structural information. The purpose of this paper is to propose conceptual and software support for the design of abstract domains. It contains two main contributions: the notion of open product and a generic pattern domain. The *open product* is a new, language-independent, way of combining abstract domains allowing each combined domain to benefit from information from the other components through the notions of queries and open operations. It provides a framework to approximate Cousots' reduced product, while reusing existing implementations and providing methodological guidance on how to build domains for interaction and composition. It is orthogonal and complementary to Granger's product which improves the direct product by a decreasing iteration sequence based on refinements but lets the domains interact only after the individual operations. *The generic pattern domain* `Pat`($\Re$) automatically upgrades a domain `D` with structural information yielding a more accurate domain `Pat(D)` without additional design or implementation cost. The two contributions are orthogonal and can be combined in various ways to obtain sophisticated domains while imposing minimal requirements on the designer. Both contributions are characterized theoretically and experimentally and were used to design very complex abstract domains such as `PAT(OPos`$\otimes$`OMode`$\otimes$`OPS)` which would be very difficult to design otherwise. On this last domain, designers need only contribute about 20% (about 3,400 lines) of the complete system (about 17,700 lines).

Keywords: Abstract Interpretation, Logic Programming, Combination of Domains

Correspondence Address:
Agostino Cortesi
Università Ca' Foscari di Venezia
Dipartimento di Informatica
via Torino 155
30170 Mestre-Venezia
ITALY
e-mail: cortesi@dsi.unive.it

2 Introduction

Abstract interpretation [16] is a systematic methodology to develop static program analysis. A traditional approach to abstract interpretation consists mainly of three steps: (1) the definition of a fixpoint semantics of the programming language: the concrete semantics; (2) the abstraction of the concrete semantics: the abstract semantics; (3) the design of a fixpoint algorithm to compute the least fixpoint of the abstract semantics. In general, the abstract semantics and the fixpoint algorithm are generic, i.e. they are parameterized by an abstract domain and its associated operations. A static analysis is then obtained by defining an abstract domain and providing an implementation of the operations as consistent approximations of the concrete operations. The main advantage of the approach is to factor out the abstract semantics and the fixpoint algorithm for various applications, providing modularity and reusability.

Abstract interpretation has raised much interest in logic programming because of the need for optimizations in compilers to make them competitive with procedural languages, the variety of interdependent analyses that need to be performed, and their sophistications which require methodological and software support. The use of abstract interpretation has led to dramatic improvements in Prolog compiler technology [54, 60]. Moreover, substantial work (e.g. [20, 4, 28, 26, 34, 24, 35]) has been devoted to produce efficient generic fixpoint algorithms and systems like `GAIA`[1] [35] have been shown to yield efficient and accurate analyses.

With the emergence of these generic fixpoint algorithms, most of the burden in developing an analysis lies in the design of the abstract domain and its associated operations. The design of abstract domains is often complex and error-prone because of the variety of interdependent analyses (e.g. freeness, sharing, groundness, types) that must be integrated, the necessity of handling structural information (i.e. information on the structure of the terms such as the functor and the arguments) to achieve reasonable accuracy, and the desire to obtain a good tradeoff between accuracy and efficiency. Yet little research has addressed the important problem of supporting this task adequately. Notable exceptions are [17, 18, 8, 25, 26, 50].

The purpose of this paper is to propose some conceptual and software support to build sophisticated abstract domains. It contains two main contributions: (1) a new product operation: the open product; (2) a generic pattern domain $\mathtt{Pat}(\Re)$ for structural information. Both contributions aim at simplifying the combination of abstract domains in practice, although their semantic foundations are also studied in detail.

The *open product* construct is a novel way of combining abstract domains, independent from logic programming and hence applicable to other programming languages as well. The key idea is the notion of open abstract domain which contains queries (providing information to the environment) and open operations (receiving information from the environment). The open product improves on the *direct product* [17] by letting the domains interact, since operations in one domain can use queries in other domains. Its formal characterization provides us with a precise meaning of consistent approximation in this open context and an automatic way of combining operations and queries. The open product provides a rich framework to express combinations of domains where the components interact, yielding what is called an attribute-dependent analysis [18]. It can be used as a way to implement or approximate the *reduced product* designed by the Cousots [17, 18] which is the most precise refinement of the direct product but may require a *revision of the original design phase* [17]. The open product is orthogonal and complementary to Granger's product which

[1] GAIA is available by anonymous ftp from Brown University.

also provides an approximation of the reduced product (with reasonable implementation effort) through a decreasing iteration sequence of refinements. However, in Granger's product, the domains only interact after the operations which may lead to a loss of precision. For instance, in logic programming, linearity information can be used in set-sharing analysis to avoid unnecessary set-closures during abstract unification [14] and it is easy to see the same improvement cannot be obtained by applying Granger's product. The open product also contains as a degenerated case the refinement operation proposed independently by [8]. It also shares some of the motivations behind the ideas of $\mathcal{R}$-abstraction of [13] and open semantics [3], although the technical details and practical applications are fundamentally different.

The *generic pattern domain* `Pat(`$\Re$`)` is more tailored to logic programming, although its principles are general and could be used for other programming languages as well. Contrary to the open product construct that is fully generic, `Pat(`$\Re$`)` is semi-generic in the sense that it combines a specific domain with an arbitrary domain. The specific domain of `Pat(`$\Re$`)` was motivated by the fact that information on the structure of terms (i.e. main functor and, recursively, structural information on subterms) dramatically improves the precision of the abstract domain albeit at a significant increase in complexity of the domain. Its main contribution amounts to upgrading automatically a domain `D` to obtain a new domain `Pat(D)` augmenting `D` with structural information. As a consequence, it provides the additional accuracy without increasing the design complexity which is factored out in `Pat(`$\Re$`)`. In addition, it makes it possible to let domains interact at a finer granularity. The key technical idea behind `Pat(`$\Re$`)` is to provide a generic implementation of the abstract operations of `Pat(D)` in terms of a few basic operations on the domain `D` using the notion of subterm that was also the basis of the pattern domain of [48, 35]. Note also that the motivations behind `Pat(`$\Re$`)` are similar to those of [26] which proposes an engine preserving structural information. One of the fundamental differences between these two approaches is that our approach handles structural information at the domain level and not inside the fixpoint algorithm. As a consequence, the domain can be combined with a variety of fixpoint algorithms achieving various tradeoffs between efficiency and accuracy.

The two contributions are completely orthogonal and can be combined in various ways to obtain sophisticated abstract domains. The main advantages of this approach are the simplicity, modularity, and accuracy it offers to abstract domain designers. *Simplicity* is achieved by abstracting away structural information and allowing designers to focus on one domain at a time. *Modularity* comes from the fact that abstract domains can be viewed as abstract data types simplifying both the correctness proofs and the implementation. Finally, *accuracy* results from structural information and from the idea of open operation which is so general that abstract domains can interact at will although through well defined interfaces.

To demonstrate the practicability of this approach, both contributions have been implemented on a large collection of abstract domains including `Pat(Pos)`, `Pat(Types)`, `Pat(OMode`$\otimes$`OPS)` and `Pat(OPos`$\otimes$`OMode`$\otimes$`OPS)`, where `Pos` is the groundness domain of Marriott and Sondergaard [42, 12, 39], `Type` is the type graph domain of Bruynooghe and Janssens [5], and `OMode` and `OPS` are well-known domains for modes and sharing. It is interesting to note that `Pat(OPos`$\otimes$`OMode`$\otimes$`OPS)` and `Pat(Type)` are some of the most complicated domains ever implemented for Prolog, yet their requirements on the designer are minimal. These experimental results clearly indicate the conceptual contributions of this paper greatly simplify the construction of complex abstract domains that are good compromises between efficiency and accuracy.

The rest of the paper is organized in four sections. Sections 3 and 4 present the main contributions of this paper, i.e. the open product and the generic pattern domain. Section 5 presents some experimental results and Section 6 concludes the paper.

3 Open Product

This section considers the problem of designing an abstract domain D as a combination of domains $D_1, \ldots, D_n$ and proposes the novel concepts of open product and refinement. Subsection 3.1 gives an overview of our approach and a comparison with some previous work in the area. Section 3.2 formalizes the concepts while Section 3.3 illustrates the approach for the abstract interpretation of Prolog. In a first reading, it may be convenient to refer to Section 3.3 when reading Section 3.2.

3.1 Overview

The *direct product* [17] is the simplest combination of abstract domains. Given two abstract domains D_1 and D_2 with concretization functions $Cc_1 : D_1 \to C$ and $Cc_2 : D_2 \to C$, the direct product domain is the domain $D = D_1 \times D_2$ with concretization function $Cc(\langle d_1, d_2\rangle) = Cc_1(d_1) \sqcap Cc_2(d_2)$. Moreover, given a concrete operation $\mathtt{OP}_c : C \to C$ and two corresponding abstract operations $\mathtt{OP}_1$ and $\mathtt{OP}_2$ on D_1 and D_2, a direct product operation can be obtained automatically as

$$\mathtt{OP}(\langle d_1, d_2\rangle) = \langle \mathtt{OP}_1(d_1), \mathtt{OP}_2(d_2)\rangle.$$

The main disadvantage of the direct product is its lack of precision since there is no interaction between the components. Note also that in general the direct product domain does not form a Galois insertion with the concrete domain. This means that it may contain redundant elements (i.e. distinct elements with the same concretization) possibly implying an additional loss of precision since the operations are not guaranteed to work on the most precise components.

The *reduced product* was proposed by the Cousots [17] to overcome some of the limitations of the direct product. Its key idea is to cluster into equivalence classes the elements of the direct product having the same concretization and to work on the more precise representative of each class. More formally, consider the function $reduce : D_1 \times D_2 \to D_1 \times D_2$ defined as

$$reduce(\langle d_1, d_2\rangle) = \bigsqcap\{\langle e_1, e_2\rangle \mid Cc(\langle e_1, e_2\rangle) = Cc(\langle d_1, d_2\rangle)\}.$$

The reduced product domain is the domain $D = \{reduce(\langle d_1, d_2\rangle) \mid d_1 \in D_1 \wedge d_2 \in D_2\}$. Clearly, the reduced product is the most precise refinement of the cartesian product. It removes redundancies from the domain and enjoys some nice theoretical properties (e.g. the reduced product of two Galois connections is a Galois connection[2]). It is also possible to specify optimal abstract operations for the reduced domain. If α_i is an abstraction function for D_i $(1 \leq i \leq 2)$, an optimal abstraction of a concrete operation $\mathtt{OP}_c$ can be specified as

$$\mathtt{OP}(\langle d_1, d_2\rangle) = reduce(\langle \alpha_1(r), \alpha_2(r)\rangle) \qquad \text{where } r = \mathtt{OP}_c(Cc(\langle d_1, d_2\rangle)).$$

However, this definition is a theoretical concept, since it uses the (non-computable) concretization functions. Moreover, as pointed out by the Cousots in [17], the implementation of abstract operations along this specification "would necessitate the revision of the original design phase". This

[2]Note that the reduced product may not apply if the domains are not Galois connections, since the greatest lower bound may not be in the same equivalence class.

of course defeat our main goal of reusing existing domains and implementations and of providing a framework for defining cooperating abstract domains.

Several papers (e.g., [8, 9]) in fact refer to a simpler version of the reduced product, called the *pseudo-reduced product* in this paper. In the pseudo-reduced product, the domain remains the same as in the direct product but the abstract operations are defined as

$$\mathtt{OP}(\langle d_1, d_2\rangle) = \mathit{reduce}(\langle \mathtt{OP}_1(d_1), \mathtt{OP}_2(d_2)\rangle).$$

In the pseudo-reduced product, the abstract operations are, in general, non-optimal However, this proposal also has some inherent limitations. On one hand, the definition still relies on the concretization function (a semantic notion). On the other hand, additional accuracy can be obtained by defining new operations where the operations on D_1 and D_2 interact.

An elegant solution to the first problem (i.e., to compute a good approximation of the *reduce* function) was proposed by Granger in [25]. The key idea is to define two new operations σ_1 and σ_2 on the product $D_1 \times D_2$ that "reduces" each of the components respectively and to iterate the application of these two operations. More precisely, let $\sigma_1 : D_1 \times D_2 \to D_1$ such that $\sigma_1(d_1, d_2) \leq d_1$ and $Cc(\langle \sigma_1(d_1, d_2), d_2\rangle) = Cc(\langle d_1, d_2\rangle)$ and let $\sigma_2 : D_1 \times D_2 \to D_2$ such that $\sigma_2(d_1, d_2) \leq d_2$ and $Cc(\langle d_1, \sigma_2(d_1, d_2)\rangle) = Cc(\langle d_1, d_2\rangle)$. *Granger's product* is defined as the fixpoint of a decreasing iteration sequence $(\langle \eta_1^n, \eta_2^n\rangle)_{n\in I\!N}$ defined as follows:

$$\begin{array}{lcl} \langle \eta_1^0, \eta_2^0\rangle & = & \langle d_1, d_2\rangle \\ \langle \eta_1^{n+1}, \eta_2^{n+1}\rangle & = & \langle \sigma_1(\eta_1^n, \eta_2^n), \sigma_2(\eta_1^n, \eta_2^n)\rangle. \end{array}$$

The main benefit of this approach is that the designer must only implement (an approximation of) the σ_i functions relating D_i with the cartesian product $D_1 \times D_2$. As a consequence, Granger's approach imposes reasonable effort and reuses the existing implementation. There are however some limitations to this proposal. The first limitation is that the domains only interact after completion of the individual operations; letting them interact *during* the individual operations may lead to additional accuracy that cannot be recovered by Granger's product. For instance, in logic programming, linearity information can be used in set-sharing analysis to avoid unnecessary set-closures during abstract unification [14] and it is easy to see this improvement cannot be obtained by applying Granger's product after the operation. The second limitation is the fact that the new operations are defined in terms of the product, which we would like to avoid in order to provide a fully automated combination of domains, to guide the design phase, and to hide irrelevant implementation details.

The *open product* is an attempt to remedy these two limitations. More precisely, the open product aims at providing a framework to

1. implement more precise approximations of the reduced product by letting domains interact during and after the operations;

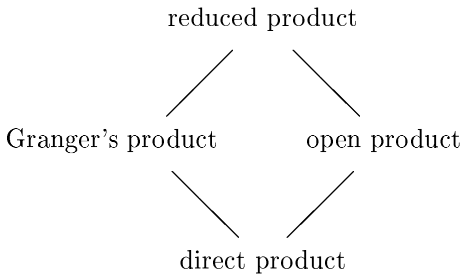
2. provide methodological guidance on how to construct abstract domains that lead to effective and precise combinations;

3. support an encapsulation of the representation and implementation of each component, thus avoiding operations that manipulate several domains simultaneously.

It is important to stress that the open product is in fact orthogonal to Granger's product and we show how to combine the two proposals by defining a version of the open product incorporating Granger's idea of refinement. It is also orthogonal to other systematic methods to build abstract domains such as down-set completion and tensor products [17, 50, 18] or the pattern domain defined in the next section.

The key idea behind the open product is the notion of open abstract interpretation.[3] Informally speaking, an open abstract interpretation differs from a traditional abstract interpretation by introducing the notion of queries and open operations. A query is simply a function giving information about the properties captured by the domain. An open operation is essentially an abstract operation except that it receives one or more boolean functions describing additional properties of the concrete objects (e.g., properties not captured by the domain). The main benefit of open interpretations is the fact that abstract operations are able to receive information from the environment to improve their accuracy. Once open interpretations are defined, it is natural to define a new form of product, the open product, which is similar to the direct product except that the open operations in one domain can use some queries in other domains to improve accuracy.

As mentioned, the open product can be improved further by letting the subdomains refine each other, by applying a variant of Granger's product. Refinements are open operations which do not modify the global concretization function but improve the subdomains locally. Refinements are orthogonal to the open product and lead to the notion of refined open product which is an open product followed by a sequence of refinements. Observe that, when operations interact just through the refinement operations, an implementation of the pseudo-reduced product is obtained. The following figure depicts the relations among product definitions with respect to the accuracy of the operations, the reduced product being the most precise and the direct product the least accurate.



It is important to stress that the notions of open product and open interpretation are both theoretical and practical tools. On the theoretical side, they capture precisely the properties that need to be satisfied to obtain a new domain and consistent operations. On the practical side, they allow the designer to build a complex domain as a set of open domains which are nothing else than abstract data types offering queries and open operations. Moreover, there exist systematic ways of composing queries from different domains and to complement incomplete interpretations. In addition, the open product provides some methodological guidance on how to build domains that are well-suited for composition and interaction. The key idea here is the recognition that abstract

[3] We are using the term "abstract interpretation" in a technical sense here as in [13].

operations are often based on some abstract properties (e.g., for instance, abstract unification uses abstract sharing in logic programming). Hence, to build complex abstract domains in a systematic way, it is important to isolate these properties into queries so that the domain can be combined with a variety of other domains supplying the abstract properties. Finally, it is important to note that the open product is completely independent of logic programming and can be used for any programming language.

3.2 Formalization

In the following we assume familiarity with standard notions of abstract interpretation [16]. We assume for simplicity that all complete partial orders (cpo) are pointed and use the following definitions for domains and abstractions of domains.

Definition 1 (domain) *A domain is a cpo, with a given upper bound (union) operation (either its lub operation or an approximation of the lub).*

Definition 2 (abstraction of domains) *Let D_1, D_2 be two domains ordered by $\leq_1$ and $\leq_2$ respectively. The domain D_2 abstracts D_1 if there exists a monotone function $Cc : D_2 \to D_1$ such that $\forall d_1 \in D_1 \; \exists d_2 \in D_2 : d_1 \leq_1 Cc(d_2)$. The function Cc is called a "concretization function".*

Additional structure can be imposed on the domains and the abstractions but this issue is orthogonal to our objectives. We also denote by $Bool$ the set $\{\mathtt{true}, \mathtt{false}\}$ and assume, without loss of generality, the order induced by $\mathtt{true} \leq \mathtt{false}$ on $Bool$. It is natural to use $\Leftarrow$ for this order. Let Arg denote a generic set, whose role will be discussed below. The first important concept we introduce is the notion of query which gives information about the properties of concrete objects.

Definition 3 (test) *Let Arg be a set. A test is a boolean function $\mathcal{T} : Arg \to Bool$. The tests on the same set Arg, denoted $Test_{Arg}$, can be partially ordered as follows: $\mathcal{T} \leq \mathcal{T}' \Leftrightarrow \forall h \in Arg : \mathcal{T}(h) \Leftarrow \mathcal{T}'(h)$.*

Definition 4 (query) *A query on the domain D wrt a given set Arg is a monotone function $\mathcal{Q} : D \to Test_{Arg}$ which maps elements of the domain D onto tests.*

The elements in Arg can be understood as "selectors" of information in an object from a domain. For the concrete domain, the information is fully precise. For an abstract domain, the information is in general an approximation. The "nature" of Arg depends basically on commonalities to the (abstract and concrete) domains considered. For instance, in the case of logic programming we may have very different domains that represent (not necessarily just) groundness information on program variables (e.g. `Pos` [42], `Mode` [48], `ASub` [7], `Sharing` [30]). Although different in shape (e.g, elements of `Pos` are propositional formulas, whereas elements of `Sharing` are sets of sets of variables), all these domains may give the environment an answer to the following test: "is a given variable x surely bound to a ground term?". Therefore, all these domain may provide an implementation of a groundness query, that is just the mentioned test applied to a particular state (i.e. to an element of the domain). In this case, the test domain Arg is just $\wp(V)$, where V is the set of program variables. Such a query may be used, for instance, whenever one of the mentioned domains is combined with another domain that may benefit from groundness information. For instance, any domain either for sharing, or freeness, or type, or compoundness analysis may make use of this environment information to specialize abstract unification, improving either in accuracy and/or in efficiency.

Alternatively, Arg can contain pairs of variables, and the test can provide information relating the terms bound to the variables such as possible sharing, equality, and covering. These queries may provide useful information, for instance, to improve the accuracy of type analysis, in a suitable combination of domains.

Queries give rise to the notion of query interpretations which is a slight generalization of the traditional notion of interpretation [13] and was proposed independently in [47] for other purposes. In the following we denote by $\mathbf{D}$ the tuple $(D, \leq, \langle \mathtt{OP}_1, \ldots, \mathtt{OP}_n \rangle, \langle \mathcal{Q}_1, \ldots, \mathcal{Q}_m \rangle)$, and by $\mathbf{D}_h$ the tuple $(D_h, \leq_h, \langle \mathtt{OP}_1^h, \ldots, \mathtt{OP}_n^h \rangle, \langle \mathcal{Q}_1^h, \ldots, \mathcal{Q}_m^h \rangle)$, for $h = 0, 1, 2$.

Definition 5 (query interpretation) *Let $\{Arg_j\}_{j \in J}$ be a finite family of sets. A query interpretation wrt $\{Arg_j\}_{j \in J}$ is a tuple $\mathbf{D}$ where: D is a domain; $\leq$ is the partial order on D; $\mathtt{OP}_1, \ldots, \mathtt{OP}_n$ are operations of signature $\mathtt{OP}_i : D \to D$;[4] $\mathcal{Q}_1 \ldots, \mathcal{Q}_m$ are queries on D wrt $Arg_{j_1}, \ldots, Arg_{j_m}$, with $j_i \in J$.*

A query interpretation can be seen as an abstract data type, where queries represent information about the domain D which is offered to the environment. Observe that, when $m = 0$ (no query), we get the traditional definition of abstract interpretation. In order to use queries, we introduce the concepts of open operation (an operation parametrized by tests) and open interpretation (an interpretation with open operations and queries).

Definition 6 (open operation) *Let $\{Arg_j\}_{j \in J}$ be a finite family of sets. An open operation on the domain D is a function $\mathtt{OP} : \langle Arg_j \to Bool \rangle_{j \in I \subseteq J} \to D \to D$ ($|I| \geq 0$) which maps a tuple of tests onto an operation. $\mathtt{OP}$ should be monotone with respect to the tests. $\mathtt{OP}$ is said open with respect to the family $\{Arg_j\}_{j \in J}$.*

In practice, tests have different signatures. For simplicity, we assume that all signatures are the same hereafter, i.e. that all queries and tests are defined wrt to a given set Arg. Moreover, in the definition above, when $|I| = 0$ we get the traditional notion of (abstract) operation.

Definition 7 (open interpretation) *An open interpretation (wrt Arg) is a tuple $\mathbf{D}$ where: D is a domain; $\leq$ is the partial order on D; $\mathtt{OP}_1, \ldots, \mathtt{OP}_n$ are open operations on D wrt Arg; and $\mathcal{Q}_1 \ldots, \mathcal{Q}_m$ are queries on D wrt Arg.*

Observe that a query interpretation can be seen as a degenerate case of open interpretation where none of the operations depends on tests. We now consider the abstraction of query interpretations and of open interpretations as generalizations of the traditional notions.

Definition 8 (abstraction of queries) *Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be two queries wrt the same set Arg on domains D_1 and D_2 respectively, and assume that D_2 abstracts D_1. We say that $\mathcal{Q}_2$ is an abstraction of $\mathcal{Q}_1$ if $\forall d \in D_2 : \mathcal{Q}_1(Cc(d)) \leq \mathcal{Q}_2(d)$, with respect to the ordering on tests.*

[4] We restrict our attention to unary operations. A generalization of the definition is straightforward.

Definition 9 (abstraction of query interpretations) *Consider the query interpretations* $\mathbf{D}_1$ *and* $\mathbf{D}_2$*. We say that* $\mathbf{D}_2$ *is an abstraction of* $\mathbf{D}_1$ *if:*

- D_2 *abstracts* D_1*;*
- *for* $1 \leq i \leq n$ *:* $\mathtt{OP}_i^2$ *abstracts* $\mathtt{OP}_i^1$*, i.e.* $\forall d_2 \in D_2 : \ \mathtt{OP}_i^1(Cc(d_2)) \leq_1 Cc(\mathtt{OP}_i^2(d_2))$;
- *for* $1 \leq i \leq m$ *:* $\mathcal{Q}_i^2$ *abstracts* $\mathcal{Q}_i^1$*.*

We now introduce the semantics of open operations through the notion of open abstraction.

Definition 10 (open abstraction) *Consider a query interpretation* $\mathbf{D}_1$ *and an open interpretation* $\mathbf{D}_2$*. We say that* $\mathbf{D}_2$ *is an open abstraction of* $\mathbf{D}_1$ *if:*

- D_2 *abstracts* D_1*;*
- *for* $1 \leq i \leq n$ *:* $\mathtt{OP}_i^2$ *abstracts* $\mathtt{OP}_i^1$*, i.e.*

$$\forall d_2 \in D_2 \ \ \forall d_1 \in D_1 : d_1 \leq_1 Cc(d_2) \ \Rightarrow \ \mathtt{OP}_i^1(d_1) \leq_1 Cc(\mathtt{OP}_i^2(\langle \mathcal{Q}_1^1(d_1), \ldots, \mathcal{Q}_m^1(d_1)\rangle)\,(d_2)).$$

- *for* $1 \leq i \leq m$ *:* $\mathcal{Q}_i^2$ *abstracts* $\mathcal{Q}_i^1$*.*

We are now in position to define the notion of open product of domains. An important point to notice here is how the product operations and the product queries are derived automatically.

Definition 11 (open product) *Consider two open interpretations* $\mathbf{D}_1$ *and* $\mathbf{D}_2$ *with respect to the same set* Arg*. The open product* $\mathbf{D}_1 \otimes \mathbf{D}_2$ *is the query interpretation* $\mathbf{D}$ *defined as follows.*

- D *is the cartesian product* $D_1 \times D_2$*;*
- *the partial ordering* $\leq$ *is the product ordering of* $\leq_1$ *and* $\leq_2$*;*
- *the query* $\mathcal{Q}_i$ *is defined as* $\mathcal{Q}_i(d_1, d_2) = \mathcal{Q}_i^1(d_1) \vee \mathcal{Q}_i^2(d_2)$*.*
- *the operation* $\mathtt{OP}_i : (D_1 \times D_2) \rightarrow (D_1 \times D_2)$ *is defined by:*

$$\mathtt{OP}_i(d_1, d_2) = (\, \mathtt{OP}_i^1(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)\,(d_1)\ ,\ \mathtt{OP}_i^2(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)\,(d_2)\,).$$

The following theorem is a soundness result which proves that the open product of two abstractions is itself an abstraction.

Theorem 1 (consistency of the open product) *Let* $\mathbf{D}_0$ *be a query interpretation wrt* Arg*, and assume the existence of a greatest lower bound operation* $\sqcap$ *on the domain of* $\mathbf{D}_0$*.*[5]

Let $\mathbf{D}_1$ *and* $\mathbf{D}_2$ *be open interpretations (wrt* Arg*) such that* $\mathbf{D}_1$ *and* $\mathbf{D}_2$ *are open abstractions of* $\mathbf{D}_0$*. The open product* $\mathbf{D}_1 \otimes \mathbf{D}_2$ *is an abstraction of* $\mathbf{D}_0$*.*

Proof: The three conditions of Definition 9 hold.

[5]Note that, in general, the existence of $\sqcap$ is trivially ensured, since D_0 will be the concrete domain, which is a complete lattice in most of applications.

1. **Domain:** Domain $D_1 \times D_2$ abstracts D_0 through the concretization function $Cc : D_1 \times D_2 \to D_0$ defined exactly as in the direct and reduced product: $Cc(d_1, d_2) = Cc(d_1) \sqcap Cc(d_2)$. This function is monotone by composition of monotone functions.

2. **Queries:** Query $\mathcal{Q}_i$ abstracts $\mathcal{Q}_i^0$ since they apply to the same domain Arg, and $d_0 \leq_0 Cc(d_1), Cc(d_2)$ implies $\mathcal{Q}_i^0(d_0) \Leftarrow \mathcal{Q}_i^1(d_1), \mathcal{Q}_i^2(d_2)$ and thus $\mathcal{Q}_i^0(d_0) \Leftarrow \mathcal{Q}_i(d_1, d_2)$.

3. **Operations:** Operation $\mathtt{OP}_i$ abstracts $\mathtt{OP}_i^0$ since, for $1 \leq h \leq 2$ and $1 \leq j \leq m$

$$
\begin{array}{l}
d_0 \leq_0 Cc(d_1), Cc(d_2) \\
\Rightarrow\ \mathtt{OP}_i^0(d_0) \leq_0 Cc(\mathtt{OP}_i^h(\langle \mathcal{Q}_1^0(d_0), \ldots, \mathcal{Q}_m^0(d_0)\rangle)(d_h)) \quad (D_h \text{ is an open abstraction of } D_0) \\
\Rightarrow\ \mathtt{OP}_i^0(d_0) \leq_0 Cc(\mathtt{OP}_i^h(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)(d_h)) \quad (\mathcal{Q}_j \text{ is an abstraction of } \mathcal{Q}_j^0) \\
\Rightarrow\ \mathtt{OP}_i^0(d_0) \leq_0 \sqcap_{1 \leq h \leq 2}\ Cc(\mathtt{OP}_i^h(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)(d_h)) \quad (\text{by properties of } \sqcap) \\
\Rightarrow \mathtt{OP}_i^0(d_0) \leq_0 Cc(\mathtt{OP}_i(d_1, d_2)) \quad (\text{by definition of } \mathtt{OP}_i).
\end{array}
$$

Hence, $\mathtt{OP}_i^0(Cc(d_1, d_2)) \leq_0 Cc(\mathtt{OP}_i(d_1, d_2))$ (by $Cc(d_1, d_2) \leq_0 Cc(d_1), Cc(d_2)$). ■

Refined Open Product The open product enables operations on the single components to benefit from information from the other components in the state. However, the operations themselves can produce additional information that may be useful to refine the results. As mentioned previously, refinements can be used after each product operation. This idea was proposed independently, but not formalized, in [8], and it is the crucial idea in the local decreasing iteration method by Granger [25]. The definition above can be seen as a slight variant of a one-step iteration of Granger's product.

Definition 12 (refinement) *Let* $\mathbf{D}_0$, $\mathbf{D}_1$ *be respectively a query interpretation and an open interpretation (wrt Arg) such that* $\mathbf{D}_1$ *is an open abstraction of* $\mathbf{D}_0$. *An operation* $\mathtt{REFINE}$: $(Arg \to Bool)^m \to D_1 \to D_1$ *is a refinement operation of* D_1 *(with respect to* D_0*) if for all* $d_0 \in D_0$ *and* $d_1 \in D_1$, *the following conditions hold.*

- $d_0 \leq_0 Cc(d_1) \ \Rightarrow\ d_0 \leq_0 Cc(\mathtt{REFINE}(\langle \mathcal{Q}_1^0(d_0), \ldots, \mathcal{Q}_m^0(d_0)\rangle)(d_1))$;
- $\mathtt{REFINE}(\langle \mathcal{T}_1, \ldots, \mathcal{T}_m\rangle)(d_1) \leq_1 d_1$ *for any test* $\mathcal{T}_1, \ldots, \mathcal{T}_m \in Test_{Arg}$.

Note that refinements are not required to be monotone[6]. Monotonicity is of course important to obtain a decreasing iteration sequences of refinements as in Granger's method.

Consider the open product $\mathbf{D}_1 \otimes \mathbf{D}_2$, where $\mathbf{D}_1$ and $\mathbf{D}_2$ are open abstractions of the query interpretation $\mathbf{D}_0$. Assume that the refinement functions $\mathtt{REFINE}_1$ and $\mathtt{REFINE}_2$ are defined in $\mathbf{D}_1$ and $\mathbf{D}_2$ respectively. The corresponding operation $\mathtt{REFINE}$ in the open product is defined as traditional operations.

Definition 13 (refinement in the open product) *In the hypotheses and notation of Theorem 1, assume that* $\mathtt{REFINE}_1$ *and* $\mathtt{REFINE}_2$ *are refinement functions for* $\mathbf{D}_1$ *and* $\mathbf{D}_2$. *The refinement function* $\mathtt{REFINE}$ *on the open product* $\mathbf{D}_1 \otimes \mathbf{D}_2$ *is defined by:*

$$\mathtt{REFINE}(\langle d_1, d_2\rangle) = (\mathtt{REFINE}_1(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)(d_1)\ ,\ \mathtt{REFINE}_2(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)(d_2))$$

[6] Monotonicity proofs may be complex and tedious for complex abstract domains and are not necessary for correctness in frameworks with monotone concretization functions

An abstract operation in the open product can now be defined as follows.

Definition 14 (refined abstract operation) *Under the hypotheses and notation of Theorem 1, assume that* $\mathtt{REFINE}_1$ *and* $\mathtt{REFINE}_2$ *are refinement functions for* $\mathbf{D}_1$ *and* $\mathbf{D}_2$.
The operation $\mathtt{OP}_i : (D_1 \times D_2) \to (D_1 \times D_2)$ *can be defined as:*

$$\begin{array}{lcl} \mathtt{OP}_i(\langle d_1, d_2 \rangle) & = & \mathtt{REFINE}(\langle d_1', d_2' \rangle) \\ \langle d_1', d_2' \rangle & = & \langle\, \mathtt{OP}_i^1(\langle \mathcal{Q}_1(d_1,d_2), \ldots, \mathcal{Q}_m(d_1,d_2) \rangle)\,(d_1)\,,\, \mathtt{OP}_i^2(\langle \mathcal{Q}_1(d_1,d_2), \ldots, \mathcal{Q}_m(d_1,d_2) \rangle)\,(d_2)\,\rangle \end{array}$$

It is easy to adapt the correctness proof to refined abstract operations. Observe that the implementation of the operations REFINE can be expressed simply by using queries. This guarantees once again the complete modularity of the approach, since the interpretations can be constructed independently.

Of course, when refinements are monotone, it is possible to apply them arbitrarily often in order to obtain additional accuracy as in Granger's method. Once again, support could be provided to automate this step when appropriate.

3.3 Application

In this section, we illustrate the refined open product to compose two domains for logic programming: a groundness and a sharing domain. We describe respectively the concrete semantics, the abstract domains and the open product. In the following, variables are taken from the set $V = \{x_1, x_2, \ldots, x_i, \ldots\}$ and we use F to denote a finite subset of V. The presentation is intentionally simplified.

3.3.1 Concrete Domain

Domain A traditional concrete domain for logic programming has sets of substitutions as elements. Given *Subst* the set of all substitutions, we denote by PS_F the set of substitutions whose domain is F. A substitution $\theta \in PS_F$ can be identified with the tuple $\langle x_{i_1}\theta, \ldots, x_{i_k}\theta \rangle$ where $x_{i_1}, \ldots, x_{i_k}$ are the elements of F. A concrete domain CS_F is simply $\wp(PS_F)$. This domain is a complete lattice with respect to the set inclusion $\subseteq$.[7]

Operations The operations on the concrete domains vary from one framework to another. However, they need to contain at least projection, unification, and an upper bound operation. In the following, for illustration purposes, we consider only a single operation, the unification of two variables, whose specification is as follows ($\Theta \in CS_F$):

$$\mathcal{C}\text{-}\mathtt{UNIF}(\Theta, x_i, x_j) = \{\theta\sigma \mid \theta \in \Theta \;\&\; \sigma \in mgu(x_i\theta, x_j\theta)\}.$$

Queries For simplicity, we consider only two queries, $\mathcal{C}-\mathtt{GROUND} : CS_F \to F \to Bool$ and $\mathcal{C}-\mathtt{NOSHARING} : CS_F \to F \times F \to Bool$, which provide information on groundness and sharing

[7] In the following, the notion of substitution composition is slightly non-standard and makes sure that the domains of $\theta\sigma$ and θ are the same set F.

and are specified as follows:

$$\mathcal{C}\text{-}\mathtt{GROUND}(\Theta)(x_i) = \begin{cases} \mathtt{true} & \text{if } \forall\theta \in \Theta : x_i\theta \text{ is a ground term} \\ \mathtt{false} & \text{otherwise} \end{cases}$$

$$\mathcal{C}\text{-}\mathtt{NOSHARING}(\Theta)(x_i, x_j) = \begin{cases} \mathtt{true} & \text{if } \forall\theta \in \Theta : x_i\theta, x_j\theta \text{ do not share variables} \\ \mathtt{false} & \text{otherwise} \end{cases}$$

3.3.2 The Open Abstract Interpretation OPos

We now turn to the first abstract domain and specifies the domain, its queries, operation, and refinement.

Domain The domain $\mathtt{Pos}_F$ [42, 12, 39] is the poset of Boolean functions that can be represented by propositional formulas constructed from F and the logical connectives $\vee, \wedge, \leftrightarrow$ and ordered by implication. The lattice is completed with the Boolean truth values. A truth assignment over F is a function $V : F \to Bool$. The value of a Boolean function f wrt a truth assignment V is denoted $V(f)$. The basic intuition behind the domain $\mathtt{Pos}_F$ is that a substitution θ is abstracted by a Boolean function f over F iff, for all instances θ' of θ, the truth assignment V defined by "$V(x_i)$ = true iff θ' grounds x_i $(1 \leq i \leq n)$" satisfies f. For instance, let $F = \{x_1, x_2\}$, $x_1 \leftrightarrow x_2$ abstracts the substitutions $\{x_1/x_3, x_2/x_3\}$, $\{x_1/a, x_2/a\}$, but neither $\{x_1/a, x_2/x_3\}$ nor $\{x_1/x_3, x_2/x_4\}$.

The concretization function for $\mathtt{Pos}_F$ is a function $Cc : \mathtt{Pos_F} \to CS_F$ defined as follows:

$$Cc(f) = \{\theta \in PS_F \mid \forall\sigma \in PS : (assign\ (\theta\sigma))(f) = true\}$$

where $assign : CS_F \to F \to Bool$ is defined by $assign\ \theta\ x_i = true$ iff θ grounds x_i.

Queries In $\mathtt{Pos}_F$, the queries are abstracted by the functions $\mathtt{OPos\text{-}GROUND}$: $\mathtt{Pos}_F \to F \to Bool$ and $\mathtt{OPos\text{-}NOSHARING}$: $\mathtt{Pos}_F \to F \times F \to Bool$ whose definitions are as follows

$$\begin{array}{lcl} \mathtt{OPos\text{-}GROUND}(f)(x_i) & \Leftrightarrow & (f \to x_i) \\ \mathtt{OPos\text{-}NOSHARING}(f)(x_i, x_j) & \Leftrightarrow & (f \to x_i) \ \vee \ (f \to x_j) \end{array}$$

Operation The unification can be abstracted as

$$\mathtt{OPos\text{-}UNIF(GROUND)}(f, x_i, x_j) = \begin{cases} f \wedge x_i \wedge x_j & \text{if } \mathtt{GROUND}(x_i) \vee \mathtt{GROUND}(x_j); \\ f \wedge (x_i \leftrightarrow x_j) & \text{otherwise .} \end{cases}$$

Observe that the definition above relies on the test GROUND, that may be answered by any query available in the environment. In other words, both Pos and any domain D combined with Pos may contribute to the precision of OPos-UNIF by supporting a query D-GROUND abstracting $\mathcal{C}$-GROUND.

Refinement The refinement in $\mathtt{Pos}_F$ is simply the function $\mathtt{OPos\text{-}REFINE(GROUND)}(f) = f \wedge x_{i_1} \wedge \ldots \wedge x_{i_p}$, where $\{x_{i_1}, \ldots, x_{i_p}\} = \{x_i \in F \mid \mathtt{GROUND}(x_i)\}$.

3.3.3 The Open Abstract Interpretation OPS

Domain The abstract domain OPS (inspired by the sharing component described in [35]) specifies the possible pair-sharing of variables between terms. The elements of $\mathtt{OPS}_F$ are binary and symmetrical relations $ps : F \times F$. The intuition is that the terms bound to x_i and x_j may share variables only when $ps(x_i, x_i)$ is true. The ordering between two abstract elements ps_1, ps_2 is defined as follows: $ps_1 \leq ps_2$ if $\forall(i,j) : ps_1(x_i, x_j) \Rightarrow ps_2(x_i, x_j)$. The concretization function $Cc : \mathtt{OPS}_F \rightarrow CS_F$ is

$$Cc(ps) = \{\theta \mid \forall x_i, x_j \in F : \ var(x_i\theta) \cap var(x_j\theta) \neq \emptyset \Rightarrow ps(x_i, x_j)\}.$$

Queries OPS supports the sharing query: $\mathtt{OPS\text{-}NOSHARING}(ps)(x_i, x_j) \ \Leftrightarrow \ (x_i, x_j) \notin ps$ and the ground query $\mathtt{OPS\text{-}GROUND}(ps)(x_i) \ \Leftrightarrow \ (x_i, x_i) \notin ps$.

Operation The unification is abstracted as $\mathtt{OPS\text{-}UNIF(GROUND)}(ps, x_i, x_j) = \tilde{ps'}$, where

$$ps' = \begin{cases} ps \setminus \{(x_k, x_l) \ \mid k \ \in \{i,j\} \vee l \in \{i,j\}\} & \text{if } \mathtt{GROUND}(x_i) \vee \mathtt{GROUND}(x_j) \\ ps \cup \{(x_i, x_j)\} \cup & \\ \{(x_k, x_l) \mid \exists k', l' : ps(x_{k'}, x_k) \,\&\, ps(x_{l'}, x_l) \,\&\, k', l' \in \{i,j\}\} \cup & \\ \{(x_i, x_k) \mid ps(x_j, x_k)\} \cup \{(x_j, x_k) \mid ps(x_i, x_k)\} & \text{otherwise.} \end{cases}$$

where $\tilde{ps}$ denotes the symmetrical closure of ps.

Refinement The refinement exploits groundness information. Let $W = \{x_i \in F \mid \ \mathtt{GROUND}(x_i) \ \}$. $\mathtt{OPS\text{-}REFINE(GROUND)}(ps) = \{(x_i, x_j) \mid ps(x_i, x_j) \,\&\, x_i \notin W \,\&\, x_j \notin W\}$.

3.3.4 The Open Product $\mathtt{OPos} \otimes \mathtt{OPS}$

The open product $\mathcal{D} = \mathtt{OPos} \otimes \mathtt{OPS} = (D, \leq, \mathcal{D}\text{-}\mathtt{REFINE}, \mathcal{D}\text{-}\mathtt{UNIF}, \langle \mathcal{D}\text{-}\mathtt{GROUND}, \mathcal{D}\text{-}\mathtt{NOSHARING}\rangle)$ can now be defined as follows:

- D is the cartesian product of the two domains and the partial order $\leq$ is $(\rightarrow, \subseteq)$.
- The queries are: $\begin{cases} \mathcal{D}\text{-}\mathtt{GROUND}(f, ps) & = & \mathtt{OPos\text{-}GROUND}(f) \vee \mathtt{OPS\text{-}GROUND}(ps) \\ \mathcal{D}\text{-}\mathtt{NOSHARING}(f, ps) & = & \mathtt{OPos\text{-}NOSHARING}(f) \vee \mathtt{OPS\text{-}NOSHARING}(ps) \end{cases}$
- The refinement is $\mathcal{D}\text{-}\mathtt{REFINE}(f, ps) = (f', ps')$ where
  $\begin{cases} f' & = & \mathtt{OPos\text{-}REFINE}(\mathcal{D}\text{-}\mathtt{GROUND}(f, ps))(f) \\ ps' & = & \mathtt{OPS\text{-}REFINE}(\mathcal{D}\text{-}\mathtt{GROUND}(f, ps))(ps) \end{cases}$
- The operation is $\mathcal{D}\text{-}\mathtt{UNIF}((f, ps), x_i, x_j) = \mathcal{D}\text{-}\mathtt{REFINE}(f', ps')$ where
  $\begin{cases} f' & = & \mathtt{OPos\text{-}UNIF}(\mathcal{D}\text{-}\mathtt{GROUND}(f, ps))(f, x_i, x_j) \\ ps' & = & \mathtt{OPS{-}UNIF}(\mathcal{D}\text{-}\mathtt{GROUND}(f, ps))(ps, x_i, x_j) \end{cases}$

Example 1 *Consider the pair $\langle f, ps \rangle \in D$, with $f = x_1 \wedge ((x_2 \vee x_3) \rightarrow x_4)$, and $ps = \{(x_2, x_4), (x_4, x_2), (x_3, x_4), (x_4, x_3)\}$, and consider the abstract unification applied to the equation $x_1 = x_3$. We get $\mathtt{OPS\text{-}UNIF(GROUND)}(ps, x_1, x_3) = \{(x_2, x_4), (x_4, x_2)\}$, and $\mathtt{OPos\text{-}UNIF(GROUND)}(f, x_3, x_4) = x_1 \wedge x_3 \wedge x_4$. Then, the refinement yields to the pair $\langle x_1 \wedge x_3 \wedge x_4, \emptyset \rangle$.*

4 The Generic Pattern Domain Pat($\Re$)

The purpose of this section is to present the second contribution of this paper. Once again, we start by giving an overview of the approach. We then formalize it, show its implementation, and discuss some applications.

4.1 Overview

It is well-known that preserving structural information in abstract domains for logic programming is often of primary importance to achieve a reasonable accuracy[8]. However, abstract domains preserving structural information are often an order of magnitude more complicated to design.

In this section, we define a generic abstract domain Pat($\Re$) which automatically upgrades a domain $\Re$ with structural information. As a consequence, the approach requires the same design and programming effort as the domain $\Re$, yet it fully benefits from the availability of structural information. The price to this pay for this important functionality is a small loss of efficiency for some domains (this is quantified experimentally later on). Contrary to the open product, Pat($\Re$) is tailored to logic programming. However, approaches similar in spirit can be used for other programming languages as well.

The key intuition behind Pat($\Re$) is to represent information on some subterms occurring in a substitution instead of information on terms bound to variables only. More precisely, Pat($\Re$) may associate the following information with each considered subterm: (1) its *pattern* which specifies the main functor of the subterm (if any) and the subterms which are its arguments; (2) its *properties* which are left unspecified and are given in the domain $\Re$. A subterm is said to be a leaf iff its pattern is unspecified. In addition to the above information, each variable in the domain of the substitutions is associated with one of the subterms. Note that the domain can express that two arguments have the same value (and hence that two variables are bound together) by associating both arguments with the same subterm. This feature produces additional accuracy by avoiding decoupling terms that are equal but it also contributes in complicating the design and implementation of the domain. The new notion of *constrained mapping* aims precisely at dealing with this issue. It should be emphasized that the pattern information is optional. In theory, information on all subterms could be kept but the requirement for a finite analysis makes this impossible for almost all applications. As a consequence, the domain shares some features with the depth-k abstraction [29, 32, 33], although Pat($\Re$) does not impose a fixed depth but adjusts it dynamically through upper bound and widening operations. This idea was already used in the domain Pattern defined in [48, 35] which can be viewed as an instance of Pat($\Re$) for some specific domains.

Pat($\Re$) is thus composed of three components: a pattern component, a same value component, and a $\Re$-component. The first two components provide the skeleton which contains structural and same-value information but leaves unspecified which information is maintained on the subterms. The $\Re$-domain is the generic part which specifies this information by describing properties of a set of tuples $< t_1, \ldots, t_p >$ where $t_1, \ldots, t_p$ are terms. As a consequence, defining the $\Re$-domain amounts essentially to define a traditional domain on substitutions. The only difference is that the $\Re$-domain is an abstraction of a concrete domain whose elements are sets of tuples (of terms) instead of sets of substitutions. This difference is conceptual and does not fundamentally affect the nature or complexity of the $\Re$-operations. The implementation of the abstract operations of

[8] An alternative is to use reexecution which, in practice, simulates the presence of structural information.

`Pat(`$\Re$`)` is expressed in terms of the $\Re$-domain operations. In general, the implementations are guided by the structural information and call the $\Re$-domain operations for basic cases.

`Pat(`$\Re$`)` can be designed in two different ways, depending upon the fact that we maintain information on all terms or only on the leaves. In the rest of this paper, we adopt the first approach for simplicity, although the second approach is more efficient for many domains $\Re$. In both cases, the main difficulty in generalizing the original pattern domain is to deal properly with global information, i.e. information which is not explicit for each subterm but constrains all subterms together. For instance, in `Pos`, groundness information is not associated with each subterm but rather is given through a global boolean formula. Specific information about a term can of course be extracted from the formula but need not be represented explicitly. The handling of global information has been achieved through the introduction of a number of novel concepts (e.g. constrained mapping), a radically new implementation of some operations (e.g. `UNION` and the ordering relation), and a generalization of many others (e.g. unification).

The identification of subterms (and hence the link between the structural component and the $\Re$-domain) is a somewhat arbitrary choice. In the following, we identify the subterms with integer indices, say $1 \ldots n$ if n subterms are considered. For instance, the substitution

$$\{x_1 \leftarrow t * a \ , \ x_2 \leftarrow a \ , \ x_3 \leftarrow y_1 \setminus [\,]\}$$

will have 7 subterms. The association of indices to them could be for instance

$$\{(1, t * a), (2, t), (3, a), (4, a), (5, y_1 \setminus [\,]), (6, y_1), (7, [\,])\}.$$

The *pattern component* (possibly) assigns to an index an expression $f(i_1, \ldots, i_n)$, where f is a function symbol of arity n and $i_1, \ldots, i_n$ are indices. If it is omitted, the pattern is said to be undefined. In our example, the (most precise) pattern component will make the following associations

$$\{(1, 2 * 3), (2, t), (3, a), (4, a), (5, 6 \setminus 7), (7, [\,])\}.$$

The *same value* component, in this example, maps x_1 to 1, x_2 to 3, and x_3 to 5.

Assuming that the $\Re$-domain is intended to be the sharing domain defined in the previous section, the $\Re$-component for the above abstract substitution is a relation $ps : \{1 \ldots 7\} \times \{1 \ldots 7\}$ which is true only for (5,5), (5,6), (6,5) and (6,6). Assuming the domain `Pos`, the most precise $\Re$-component for the above abstract substitution can be expressed by the formula: $1 \wedge 2 \wedge 3 \wedge 4 \wedge (5 \leftrightarrow 6) \wedge 7$. Note the use of integers instead of the variables of the previous section. This is the only difference between the $\Re$-domain and a traditional domain.

4.2 The Abstract Domain

We now turn to the formalization of `Pat(`$\Re$`)`. In the following, we denote by I_p the set of indices $\{1, \ldots, p\}$, by ST_p the set of tuples of terms $< t_1, \ldots, t_p >$, by ST the union of all sets ST_p for some $p \geq 0$, and by $\wp(ST)$ the powerset of ST.

An abstract substitution β over the program variables $F = \{x_1, \ldots, x_n\}$ is a triple (frm, sv, ℓ) where frm is called "the pattern component", sv is called "the same value component", and ℓ is called "the $\Re$-component", where $\Re$ is a domain to be specified. We say $dom(\beta) = F$.

Pattern Component The pattern component is defined as in [35]. It associates with some of the indices in I_p a pattern $f(i_1, \ldots, i_q)$, where f is a function symbol of arity q and $\{i_1, \ldots, i_q\} \subset I_p$.

We denote by FRM_p the set of all partial functions frm for a fixed p and by FRM the union of all FRM_p $(p \geq 0)$. The meaning of an element frm is given by the concretization function $Cc : FRM_p \to \wp(ST_p)$:

$$Cc(frm) \;=\; \{< t_1, \ldots, t_p > \mid \; \forall i : 1 \leq i \leq p : frm(i) = f(i_1, \ldots, i_q) \Rightarrow t_i = f(t_{i_1}, \ldots, t_{i_q})\}.$$

Same Value Component The second component assigns a subterm to each variable in the abstract substitution. Given the set F of program variables and a set of indices I_m, this component is a (total) surjective function $sv : F \to I_m$. We denote by $SV_{F,m}$ the set of all same value functions for fixed F and m and by SV the union of all sets $SV_{F,m}$ for any F and m. The meaning of an element sv is given by a concretization function $Cc : SV_{F,m} \to CS_F$ that makes sure that two variables assigned to the same index have the same value:

$$Cc(sv) = \{\theta \mid dom(\theta) = F \text{ and } \forall x_i, x_j \in F : sv(x_i) = sv(x_j) \Rightarrow x_i\theta = x_j\theta\}.$$

The $\Re$-component The $\Re$-component of the generic domain is an element of a domain $\Re_p$ that gives information on a tuple of terms $\langle t_1, \ldots, t_p \rangle$. These objects (i.e. the elements of $\Re_p$) are called $\Re$-tuples in the following. The domain is assumed to satisfy the requirements of Definition 1. In the following, we denote by $\Re$ the union of all $\Re_p$ $(p \geq 0)$. The signature of the concretization function Cc is $Cc : \Re_p \to \wp(ST_p)$.

The $\Re$-domain should include a number of operations which differ from one framework to another. Conceptually, only three operations are needed: upper bound, unification, and constrained mapping. The first two, $\Re$-`UNION` and $\Re$-`UNIF`, are rather standard and must be consistent abstractions of the following concrete operations (Φ, Φ_1 and Φ_2 are sets of p-tuples of terms):

Upper Bound: This operation takes the union of two sets of tuples.

$$\mathcal{C}\text{-UNION}(\Phi_1, \Phi_2) \;=\; \Phi_1 \cup \Phi_2.$$

Unification: This operation performs unification and needs only consider two simple cases.[9] Let g/p denote a functor of arity p.

$$\begin{array}{lcll} \mathcal{C}\text{-UNIF}(\Phi, i, j) & = & \{\langle t_1\sigma, \ldots, t_n\sigma \rangle & \mid \; \langle t_1, \ldots, t_n \rangle \in \Phi \;\&\; \sigma \in mgu(t_i, t_j)\}; \\ \mathcal{C}\text{-UNIF}(\Phi, i, g/p) & = & \{\langle t_1\sigma, \ldots, t_n\sigma, y_1\sigma, \ldots, y_p\sigma \rangle & \mid \; \langle t_1, \ldots, t_n \rangle \in \Phi \;\&\; \\ & & & \quad \sigma \in mgu(t_i, g(y_1, \ldots, y_p)) \;\&\; \\ & & & \quad y_1, \ldots, y_p \text{ are fresh variables}\} \end{array}$$

Example 2 *We show the implementation of these operations for* `Pos`*. The upper bound in* `Pos` *is the disjunction of the two formulas and is optimal in accuracy [15].*

`Pos-UNION`$(f_1, f_2) = f_1 \vee f_2$.

The unification operations are also optimal in accuracy and amount to adding an equivalence between the unified arguments. In the second case, we assume that $i_1, \ldots, i_p$ are the new indices.

`Pos-UNIF`$(f, i, j) = f \wedge (i \leftrightarrow j)$.
`Pos-UNIF`$(f, i, g/p) = f \wedge (i \leftrightarrow (i_1 \wedge \ldots \wedge i_p))$.

[9] The other cases come for free through `Pat(`$\Re$`)`. See Section 4.4.1.

The third operation, *constrained mapping*, is novel and generalizes many operations such as projection (and thus renaming), and extension. It is motivated by one of the fundamental difficulties encountered when designing the operations of `Pat(ℜ)`: the fact that abstract substitutions may have different structures in the pattern component and that equality constraints are enforced implicitly by repeated use of the same index. As a consequence, it is non-trivial to establish a correspondence between the elements of the respective $\Re$-components of two abstract substitutions and the need for such a correspondence appears, in one form or another, in many abstract operations such as `UNION` and `INTER`[10] and the ordering relation on `Pat(ℜ)`. The constrained mapping provides a uniform solution to this problem and simplifies dramatically the implementation of many abstract operations.

Observe that operations like projection, renaming and extension are not strictly necessary to design a concrete operational semantics of logic programs, but they are crucial for abstract semantics. This is the reason why the definition of the constrained mapping operation is given just at the abstract level.

Definition 15 (Constrained Mapping) *A* constrained mapping *on domain $\Re$ maps any function $tr : I_{p_2} \rightarrow I_{p_1}$ onto a function $tr^{\#} : \Re_{p_1} \rightarrow \Re_{p_2}$. This mapping has to satisfy the following conditions:*

1. $id^{\#}_{I_p}(\ell) = \ell$ *where* id_{I_p} *is the identity function on* I_p*;*

2. $(tr_1 \circ tr_2)^{\#} \leq tr^{\#}_2 \circ tr^{\#}_1$*;*

3. $\ell \leq_{\Re_{p_1}} \ell'$ *implies* $tr^{\#}(\ell) \leq_{\Re_{p_2}} tr^{\#}(\ell')$*;*

4. *(Consistency):* $\{\langle t_{tr(1)}, \ldots, t_{tr(p_2)} \rangle \ : \ \exists \langle t_1, \ldots, t_{p_1} \rangle \in Cc(l)\} \ \subseteq \ Cc(tr^{\#}(l))$.

The intuition is as follows: an element of $\Re_{p_2}$ is a constraint over the set of tuples of the form $< t_1, \ldots, t_{p_2} >$. A function $tr : I_{p_2} \rightarrow I_{p_1}$ contains two implicit pieces of information: first, a set of equality constraints for terms whose indices are mapped onto the same value by tr; second, it ignores terms whose indices are not the image of some index in I_{p_2}. This intuition is formally captured by function $tr^{\#}$ which indicates how to transform an abstract object in $\Re_{p_2}$ by removing superfluous terms and duplicating some others. Conditions 1 to 3 are useful for defining an ordering on `Pat(ℜ)`. The first condition is obvious. The second condition is a monotonicity requirement through composition. It could be sometimes transformed into an equality but this would be too strong a requirement in general. The third condition is natural: the ordering on domains must be respected, since new equal terms are added in the same way to all elements of the domain. Condition 4 is the expected consistency condition. A detailed study of these properties, and of possible alternatives, can be found in [36].

The constrained mapping can be implemented in a generic way in terms of simpler operations (see Section 4.5) demonstrating that this concept is indeed natural for many domains. More specific implementations are often simpler and more efficient but they complicate somewhat the task of the designer.

We now illustrate the constrained mapping on the abstract domains defined previously. In the case of the sharing domain, the constrained mapping implementation is much simpler than the generic implementation due to the fact that information is essentially local to pair of indices.

[10]Intersection is used for example in reexecution frameworks (e.g. [38]).

Example 3 (Constrained Mapping for the Sharing Domain.)
Let $tr : I_{p_2} \to I_{p_1}$ and $l \subseteq I_{p_1} \times I_{p_1}$ be a symmetrical relation. The constrained mapping can be defined as follows:

$$tr^{\#}(l) = \{(i,j) | (tr(i), tr(j)) \in l\}.$$

For instance, for

$$\begin{array}{rcl} l & = & \{(1,1),(7,7),(1,7),(7,1)\} \\ tr & = & \{1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 3,\ 4 \mapsto 7, 5 \mapsto 7\} \end{array}$$

the constrained mapping is

$$tr^{\#}(l) = \{(1,1),(4,4),(5,5),(1,4),(4,1),(1,5),(5,1),(4,5),(5,4)\}.$$

We show that the requirements of Definition 15 hold:

1. $id^{\#}_{I_p}(l) = \{(i,j) | (id_{I_p}(i), id_{I_p}(j)) \in l\} = \{(i,j) | (i,j) \in l\} = l.$

2.
$$\begin{array}{rcl} (tr_2 \circ tr_1)^{\#}(l) & = & \{(i,j) | ((tr_2 \circ tr_1)(i), (tr_2 \circ tr_1)(j)) \in l\} \\ & = & \{(i,j) | (tr_1(i), tr_1(j)) \in \{(i',j') | (tr_2(i'), tr_2(j')) \in l\}\} \\ & = & tr_1^{\#}(tr_2^{\#}(l)). \end{array}$$

3.
$$\begin{array}{lll} l \leq l' & & \textit{implies} \\ l(i,j) \Rightarrow l'(i,j) & \forall (i,j) \in I_{p_1} \times I_{p_1} & \textit{implies} \\ l(tr(i), tr(j)) \Rightarrow l'(tr(i), tr(j)) & \forall (i,j) \in I_{p_2} \times I_{p_2} & \textit{implies} \\ tr^{\#}(l) \leq tr^{\#}(l'). & & \end{array}$$

4.
$$\begin{array}{ll} \langle t_1, \ldots, t_{p_1} \rangle \in Cc(l) & \textit{implies} \\ \forall i,j \in I_{p_1} : var(t_i) \cap var(t_j) \neq \emptyset \Rightarrow l(i,j) & \textit{implies} \\ \forall i,j \in I_{p_2} : var(t_{tr(i)}) \cap var(t_{tr(j)}) \neq \emptyset \Rightarrow l(tr(i), tr(j)) & \textit{implies} \\ \forall i,j \in I_{p_2} : var(t_{tr(i)}) \cap var(t_{tr(j)}) \neq \emptyset \Rightarrow tr^{\#}(l)(i,j) & \textit{implies} \\ \langle t_{tr(1)}, \ldots, t_{tr(p_2)} \rangle \in Cc(tr^{\#}(l)). & \end{array}$$

For the domain `Pos`, it is difficult to give a simple "syntactical" definition of the constrained mapping in terms of Boolean formulas. The difficulty comes from the global nature of the information stored in `Pos`. As a consequence, any reasonable implementation of the constrained mapping is likely to be very close to the generic implementation described later in the paper. In the example, we give a "semantic" definition of the constrained mapping in `Pos`. The definition is not useful from an implementation standpoint but provides some insight on the nature on the constrained mapping.

Example 4 (Constrained Mapping for the Domain `Pos`) *Let the elements of $\texttt{Pos}_{I_p}$ be considered as Boolean functions of signature $(I_p \to Bool) \to Bool$ and V and W be truth assignments of signature $I_p \to Bool$. Let also $tr : I_{p_2} \to I_{p_1}$ and $l \in \texttt{Pos}_{I_{p_1}}$ (i.e. $l \in (I_{p_1} \to Bool) \to Bool$).*

The constrained mapping $tr^{\#}(l)$ can be defined as the unique function $l' \in (I_{p_2} \to Bool) \to Bool$ such that, for all $V \in I_{p_2} \to Bool$,

$$V(l') = \bigvee_{\substack{W \in I_{p_1} \to Bool \\ V = W \circ tr}} W(l).$$

For instance, assuming $p_1 = 7$ *and* $p_2 = 5$ *and the functions*

$$\begin{array}{lcl} l & = & (1 \leftrightarrow (2 \wedge 3)) \wedge (3 \leftrightarrow 7) \\ tr & = & \{1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 3,\ 4 \mapsto 7,\ 5 \mapsto 7\} \end{array}$$

$tr^{\#}(l)$ *can be expressed by the "syntactical" formula:*

$$(1 \leftrightarrow (2 \wedge 3)) \wedge (3 \leftrightarrow 4) \wedge (4 \leftrightarrow 5).$$

This comes from the semantic definition

$$V(l') = \bigvee_{\substack{W \in I_{p_1} \to Bool \\ V = W \circ tr}} (W(1) \leftrightarrow (W(2) \wedge W(3))) \wedge (W(3) \leftrightarrow W(7)).$$

and the following equalities

$$\begin{array}{lclcll} V(i) & = & W(tr(i)) & = & W(i) & (1 \leq i \leq 3) \\ V(4) & = & W(tr(4)) & = & W(7) & \\ V(5) & = & W(tr(5)) & = & W(7) & . \end{array}$$

Hence, $W(1)$*,* $W(2)$ *and* $W(3)$ *can be replaced by* $V(1)$*,* $V(2)$ *and* $V(3)$ *and* $W(7)$ *can be replaced either by* $V(4)$ *or by* $V(5)$ *in the above formula. Furthermore the constraint* $V = W \circ tr$ *can be simplified to* $V(4) = V(5)$*. It follows that*

$$\begin{array}{lcl} V(l') & = & \bigvee_{V(4)=V(5)} (V(1) \leftrightarrow (V(2) \wedge V(3))) \wedge (V(3) \leftrightarrow V(4)) \\ & = & (V(1) \leftrightarrow (V(2) \wedge V(3))) \wedge (V(3) \leftrightarrow V(4)) \wedge (V(4) \leftrightarrow V(5)). \end{array}$$

leading the obtained result.

We now show the correctness of the semantic definition.

1. $V(id^{\#}_{I_p}(l)) = \bigvee_{V = W \circ id_{I_p}} W(l) = V(l).$

2. $V(tr_2^{\#}(tr_1^{\#}(l))) = \bigvee_{V = W' \circ tr_2} W'(tr_1^{\#}(l)) = \bigvee_{V = W \circ tr_1 \circ tr_2} W(l) = V((tr_1 \circ tr_2)^{\#}(l)).$

3.

$$\begin{array}{ll} l \leq l' & implies \\ W(l) \Rightarrow W(l')\ \ \forall W \in I_{p_1} \to Bool & implies \\ \bigvee_{W \in I_{p_1} \to Bool\ \&\ V = W \circ tr} W(l) \Rightarrow \bigvee_{W \in I_{p_1} \to Bool\ \&\ V = W \circ tr} W(l')\ \ \forall V \in I_{p_2} \to Bool & implies \\ V(tr^{\#}(l)) \Rightarrow V(tr^{\#}(l'))\ \ \forall V \in I_{p_2} \to Bool & implies \\ tr^{\#}(l)\ \leq\ tr^{\#}(l'). & \end{array}$$

4.

$$\begin{array}{lll} \langle t_1, \ldots, t_{p_1} \rangle \in Cc(l) & & implies \\ assign(\langle t_1\sigma, \ldots, t_{p_1}\sigma \rangle)(l) = true & \forall\sigma & implies \\ \exists W : W(l) = true \text{ and } assign(\langle t_{tr(1)}\sigma, \ldots, t_{tr(p_2)}\sigma \rangle) = W \circ tr & \forall\sigma & implies \\ \bigvee_{assign(\langle t_{tr(1)}\sigma, \ldots, t_{tr(p_2)}\sigma \rangle) = W \circ tr} W(l) = true & \forall\sigma & implies \\ assign(\langle t_{tr(1)}\sigma, \ldots, t_{tr(p_2)}\sigma \rangle)(tr^{\#}(l)) = true & \forall\sigma & implies \\ \langle t_{tr(1)}\sigma, \ldots, t_{tr(p_2)}\sigma \rangle \in Cc(tr^{\#}(l)). & & \end{array}$$

The Domain `Pat(ℜ)` Let D be a finite set of variables. The set of abstract substitutions `Pat(ℜ)` is the subset of $FRM \times SV \times \Re$ whose elements (sv, frm, ℓ) satisfy the following conditions: 1) $\exists m,p \in N,\ p \geq m \ \&\ \ell \in \Re_p \ \&\ sv \in SV_{D,m} \ \&\ frm \in FRM_p$; 2) $\forall i : m < i \leq p : \exists j : 1 \leq j \leq p : frm(j) = f(\ldots, i, \ldots)$.

The Concretization Function Formally, the meaning of an abstract substitution $\beta = (frm, sv, \ell)$ is given by the concretization function $Cc : \texttt{Pat(ℜ)} \rightarrow CS_F$ defined by

$$Cc(\beta) = \{\theta \mid dom(\theta) = F \ \&\ \exists \langle t_1, \ldots, t_p \rangle \in Cc(\ell) \cap Cc(frm) : \forall x \in F : x\theta = t_{sv(x)}\}$$

The Ordering It remains to define the ordering relation. Consider two abstract substitutions β_1, β_2, and assume in the following that frm_i, sv_i, ℓ_i are the components of a substitution β_i, p_i is the number of indices in the domains of frm_i, and m_i is the number of indices in the codomain of sv_i[11]. Conceptually, $\beta_1 \leq \beta_2$ holds iff β_1 imposes the same or more constraints on all components than β_2 does, i.e. iff $Cc(\beta_1) \subseteq Cc(\beta_2)$. The formalization of this intuition uses the constrained mapping to establish the correspondence between the elements of the `Pat(ℜ)` domains.

Definition 16 *$\beta_1 \leq \beta_2$ iff there exists a function $tr : I_{p_2} \rightarrow I_{p_1}$ satisfying*

1. *$tr^{\#}(\ell_1) \leq_{\Re_{p_2}} \ell_2$;*
2. *$\forall x \in F : sv_1(x) = tr(sv_2(x))$;*
3. *$\forall i \in I_{p_2} : frm_2(i) = f(i_1, \ldots, i_q) \Rightarrow frm_1(tr(i)) = f(tr(i_1), \ldots, tr(i_q))$.*

Note that the above relation is only a preorder. Formally, in order to meet a Galois insertion with the concrete domain, the domain should be defined as the quotient of `Pat(ℜ)` by the equivalence relation induced by this preorder (as in the reduced domain construction). In practice, it suffices to work with arbitrary elements and hence we will continue working on the abstract domain `Pat(ℜ)`.

4.3 Implementation of the Upper Bound Operation

We turn to the implementation of the abstract operations. Operation `Pat(ℜ)-UNION` illustrates well the process of building `Pat(ℜ)` operations in terms of ℜ-operations and the benefit of the constrained mapping to overcome the difficulty encountered for certain operations in presence of global domains.

Specification 1 *Let β_1, β_2 be two abstract substitutions such that $dom(\beta_1) = dom(\beta_2) = F$.* `Pat(ℜ)-UNION`$(\beta_1, \beta_2)$ *produces an abstract substitution β such that $dom(\beta) = F \quad \& \quad \beta_1, \beta_2 \leq \beta$.*

To implement the function `Pat(ℜ)-UNION`(β_1, β_2) we need to build the set of pairs (i, j) of indices that are in correspondence. Let F be the domain of β_1 and β_2. We define the set E of pairs in correspondence induced by the same value component:

$$E = \{(i, j) \mid \exists x \in F : i = sv_1(x) \ \&\ j = sv_2(x)\}.$$

The remaining correspondences can be obtained from E and the pattern component. We define the set G of all correspondences as the smallest set satisfying

[11] The domain of sv_i is implicitly defined by the substitution.

1. $(i,j) \in E \Rightarrow (i,j) \in G$

2. $(i,j) \in E \;\&\; frm_1(i) = f(i_1,\ldots,i_n) \;\&\; frm_2(j) = f(j_1,\ldots,j_n) \Rightarrow (i_k,j_k) \in G \; (1 \leq k \leq n)$.

The number of indices in the abstract substitution produced by the `Pat(ℜ)-UNION` operation will be exactly the size of G, i.e. $p = \#G$, as these are precisely the terms corresponding in both abstract substitutions. Of course, the number of variables n is the same in β, β_1, β_2. We also need a bijective function $tr : G \to I_p$ to establish the relation between the old and the new indices of the corresponding subterms. We denote by $tr_1 : I_p \to I_{p_1}$ and $tr_2 : I_p \to I_{p_2}$ the functions mapping elements of I_p to I_{p_1} and I_{p_2} respectively. $tr_1(k) = i$ if there exists $(i,j) \in G$ such that $tr(i,j) = k$, and analogously $tr_2(k) = j$ if there exists $(i,j) \in G$ such that $tr(i,j) = k$

Implementation 1 *Operation* `Pat(ℜ)-UNION`(β_1, β_2) *produces* $\beta = (frm, sv, \ell)$*, an abstract substitution defined as follows.*

$$
\begin{array}{lcl}
frm & = & \{\langle tr(i,j), f(tr(i_1,j_1),\ldots,tr(i_n,j_n))\rangle \mid (i,j) \in G \;\&\; frm_1(i) = f(i_1,\ldots,i_n) \;\&\; \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad frm_2(j) = f(j_1,\ldots,j_n)\} \\
sv(x) & = & tr(sv_1(x), sv_2(x)) \quad \forall x \in F \\
\ell & = & \Re\texttt{-UNION}(tr_1^{\#}(\ell_1), tr_2^{\#}(\ell_2))
\end{array}
$$

Operation `Pat(ℜ)-UNION` is typical of many operations. It shows that the initial computation is driven by the pattern and the same-value components to determine how to apply the ℜ-operations. The various components are then deduced independently. Note also the simplicity gained by the availability of the constrained mapping.

Example 5 *We illustrate the upper bound operation on* `Pat(Pos)`*. In the example, the concretizations of* β_1 *and* β_2 *contain respectively the concrete substitutions* $\theta_1 = \{x_1 \leftarrow g(y_1, f(y_2,y_3)), x_2 \leftarrow g(y_4,y_4)\}$ *and* $\theta_2 = \{x_1 \leftarrow g(z_1,z_2), x_2 \leftarrow g(z_3,z_2)\}$*, respectively. More precisely, the components of* $\beta_i = (sv_i, frm_i, l_i)$ *are defined as follows.*

$$
\left\{
\begin{array}{lcl}
sv_1 & = & \{x_1 \mapsto 1, x_2 \mapsto 6\} \\
frm_1 & = & \{1 \mapsto g(2,3), 3 \mapsto f(4,5), 6 \mapsto g(7,7)\} \\
l_1 & = & (1 \leftrightarrow (2 \wedge 3)) \wedge (3 \leftrightarrow (4 \wedge 5)) \wedge (6 \leftrightarrow 7) \wedge (3 \leftrightarrow 7)
\end{array}
\right.
$$

$$
\left\{
\begin{array}{lcl}
sv_2 & = & \{x_1 \mapsto 1, x_2 \mapsto 4\} \\
frm_2 & = & \{1 \mapsto g(2,3), 4 \mapsto g(5,3)\} \\
l_2 & = & (1 \leftrightarrow (2 \wedge 3)) \wedge (4 \leftrightarrow (5 \wedge 3))
\end{array}
\right.
$$

The set G *of all the correspondences between indices induced by the same value component is*

$G = \{1 : (1,1), 2 : (2,2), 3 : (3,3), 4 : (6,4), 5 : (7,5), 6 : (7,3)\}$

The functions which establish the relations between the old and the new indices are

$$
\begin{array}{ll}
tr_1 : I_6 \to I_7; & tr_1 = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 6, 5 \mapsto 7, 6 \mapsto 7\} \\
tr_2 : I_6 \to I_5; & tr_2 = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 6 \mapsto 3\}.
\end{array}
$$

The corresponding constrained mapping are[12]

[12] The computations of the constrained mapping will be illustrated later on in Example 7

$tr_1^{\#}(l_1) = (1 \leftrightarrow (2 \wedge 3)) \wedge (4 \leftrightarrow 5) \wedge (5 \leftrightarrow 6) \wedge (3 \leftrightarrow 5).$
$tr_2^{\#}(l_2) = (1 \leftrightarrow (2 \wedge 3)) \wedge (4 \leftrightarrow (5 \wedge 3)) \wedge (3 \leftrightarrow 6)$

Therefore `Pat(ℜ)−UNION`(β_1, β_2) *returns the following abstract substitution* (frm, sv, f)*:*

$sv = \{x_1 \mapsto 1, x_2 \mapsto 4\}$
$frm = \{1 \mapsto g(2,3), 4 \mapsto g(5,6)\}$
$f = tr_1^{\#}(l_1) \vee tr_2^{\#}(l_2) = (1 \leftrightarrow (2 \wedge 3)) \wedge (4 \leftrightarrow (5 \wedge 6)) \wedge (3 \leftrightarrow 6).$

We are in position now to verify that, by construction, `Pat(ℜ)` is a domain in the sense of Definition 1.

Theorem 2 *Let* $\Re$ *be a domain. Then, the quotient of* `Pat(ℜ)` *with respect to the equivalence relation induced by the preorder* $\leq$*, enhanced with a top and bottom elements, is a domain, with* `UNION` *as selected upper bound operation.*

This paper does not study the formal properties of `Pat(ℜ)` as a domain. It can be proven that `Pat(ℜ)` is a Galois insertion if $\Re$ is a Galois connection and if the condition

$$\alpha(Cc(tr^{\#}(l))) \leq tr^{\#}(\alpha(Cc(l)))$$

holds for all abstract element l. When $\Re$ is a Galois insertion, the condition always holds but `Pat(ℜ)` is not necessarily a Galois insertion.

4.4 Implementation of the Unification

The purpose of this section is to show the implementation of the unification in `Pat(ℜ)`. We consider the operation `Pat(ℜ)-UNIF`(β, x_i, x_j) of `GAIA` which is specified as a consistent approximation of a concrete unification operation unifying the terms associated with two variables in the substitution. This operation requires the full abstract unification of `Pat(ℜ)`. In the following, by abuse of language, we often use "abstract term i" to denote the information associated with an index i in a substitution β. We also use "abstract unification of abstract terms i and j" to denote the result of an operation whose result is an abstract substitution approximating the set of concrete substitutions resulting from the unification of the terms t_i and t_j in all the substitutions belonging to $Cc(\beta)$.

4.4.1 Overview

The kernel of the unification operation in `Pat(ℜ)` is a procedure to unify two abstract terms. The abstract unification of two abstract terms i and j follows closely the concrete unification process. However, our implementation differs on certain aspects to avoid a too tedious case analysis and to simplify the task of the designer of the $\Re$-domain. Our abstract unification considers three cases depending upon the pattern components of i and j:

- i and j are leaves;

- i has a pattern $f(i_1, \ldots, i_n)$ and j has a pattern $f(j_1, \ldots, j_n)$

- i has a pattern $f(i_1, \ldots, i_n)$ and j is a leaf.

The first case is obviously the basic case and amounts mainly to applying operation $\Re$-UNIF on i and j. It is performed by operation UNIF1 below. The second case is a pure recursive case and essentially amounts to unifying the arguments recursively. The third case could be considered as a basic case but would entail substantial complication in the design of the unification operation on the $\Re$-domain. Our implementation takes an alternative approach which consists in reducing it to the second case by specializing j so that it has a pattern component. In the concrete domain, this corresponds to unify t_i and t_j in two steps:

a. unify t_j with a term $f(y_1, \ldots, y_n)$ where $y_1, \ldots, y_n$ are fresh variables, returning σ;

b. unify $f(y_1\sigma, \ldots, y_n\sigma)$ and t_i giving the desired result.

Clearly this process is equivalent to the direct unification of t_i and t_j. Step b is the second case discussed above. Step a is a unification which is much simpler in general than considering the third case as a basic case. It is performed by operation SPECAT below. Note that this approach also explains why the designer can restrict attention to the two basic cases of unification mentioned in Section 4.2.

The last point which deserves to be mentioned is the removal of abstract subterms which are no longer necessary. After unifying i and j, only one of the subterms must be preserved to improve accuracy in many cases.[13] This is achieved by operation FCTA below which replaces one index by another in the various components of the substitution. FCTA makes use of the constrained mapping for the $\Re$-component. Note, in particular, that all the new variables introduced by SPECAT are removed later on by FCTA.

The rest of this section is organized in the following way: we start by introducing some notations which significantly simplify the definitions. We then present the suboperations FCTA, UNIF1, SPECAT and UNIF. Finally, we present the Pat($\Re$) operation Pat($\Re$)-UNIF(β, x_i, x_j) and illustrate its computation on Pat(Pos).

4.4.2 Notation

The removal operations are rather frequent in the unification process and, instead of updating permanently the components, the equalities will be stored using a function $fi : I_q \to I_p$ such that $fi(i) = fi(j) \Rightarrow t_i = t_j$. This allows us to simplify the presentation and the implementation as well[14]. The idea is that the same value component needs only to be updated at the very end of each operation. So for the moment we restrict attention to two components omitting the same value component.

We call a δ-tuple the association (frm, ℓ, fi) of two components frm and ℓ defined on the same set of indices I_p and a function fi. The unification suboperations are defined on δ-tuples. Note also that we implicitly assume that a δ-tuple δ_k is associated to a tuple (frm_k, ℓ_k, fi_k) (and similarly

[13] Additional accuracy comes from the fact that the two subterms are never decoupled in this way.

[14] At the implementation level, the function fi is close to the unification process of Prolog.

a δ-tuple δ' to a tuple (frm', ℓ', fi')). As usual, we define the meaning of δ-tuples by means of a concretization function as follows:

$$\begin{array}{rcl} Cc(\delta) & = & \{(u_1, \ldots, u_q) : \exists (t_1, \ldots, t_p) \in Cc(frm) \cap Cc(\ell) : u_i = t_{fi(i)} \quad (1 \le i \le q)\} \\ & = & \{(t_{fi(1)}, \ldots, t_{fi(q)}) : (t_1, \ldots, t_p) \in Cc(frm) \cap Cc(\ell)\}. \end{array}$$

In the rest of the presentation, we will often have to write expressions such as $expr(t(i_1), \ldots, t(i_n))$ where $i_1, \ldots, i_n \in I_{p_2}$ and $t : I_{p_2} \to I_{p_1}$. We take the convention of representing those expressions as

$$expr(i_1, \ldots, i_n) \quad (t)$$

meaning that all indices i_k from I_{p_2} in the expression have to be substituted their values $t(i_k)$.

4.4.3 The Sub-Operation $\texttt{FCTA}(i, j, \delta) = \delta'$

The first operation, $\texttt{FCTA}(i, j, \delta) = \delta'$, that we define on δ-tuples amounts to adding an equality between terms t_i and t_j and to propagating this equality in the rest of tuple δ. The basic idea behind this operation is to remove a subterm which is no longer necessary.

Specification 2 *Let $\bar{u} = (u_1, \ldots, u_q)$ be a δ-tuple of terms. Then the following holds:*

$$\left.\begin{array}{l} \bar{u} \in Cc(\delta) \\ u_i = u_j \end{array}\right\} \Rightarrow \bar{u} \in Cc(\delta').$$

Let us define two functions, assuming $max = max(i, j) \quad (fi)$

1. $ti : I_p \to I_{p-1}$
$$\begin{array}{rcll} ti(k) & = & k & \text{if } k < max; \\ & = & min(i,j) \quad (fi) & \text{if } k = max; \\ & = & k - 1 & \text{if } k > max; \end{array}$$

2. $tr : I_{p-1} \to I_p$
$$\begin{array}{rcll} tr(k) & = & k & \text{if } k < max; \\ & = & k + 1 & \text{if } k \ge max. \end{array}$$

When applied to the components, the function ti removes one of the terms (the one with the largest index) by pushing leftwards the indices which are greater than the removed term while the function it allows us to retrieve previous information.

Implementation 2 *The function $\texttt{FCTA}(i, j, \delta) = \delta'$ is defined as*

$$\begin{array}{lcl} frm' & = & \{\langle ti(k), f(ti(k_1), \ldots, ti(k_n))\rangle : \langle k, f(k_1, \ldots, k_n)\rangle \in frm\}; \\ \ell' & = & tr^{\#}(\ell); \\ fi' & = & ti \circ fi. \end{array}$$

Note that the function fi of the δ-tuple needs to be updated as well. The constrained mapping is used as a projection operation here.

4.4.4 The Sub-operation $\mathtt{UNIF1}(i,j,\delta) = \delta'$

We now turn to the basic case of the unification process.

Specification 3 *This operation is defined on a δ-tuple $\delta = (frm, \ell, fi)$. It assumes that $frm(i) = frm(j) = undef$ (fi) and produces another δ-tuple $\delta' = (frm', \ell', fi')$. Informally, $\mathtt{UNIF1}(i,j,\delta)$ unifies subterms i and j in the δ-tuple δ, giving δ'.*

More formally, given a p_tuple of terms $\bar{t} = (t_1, \ldots, t_p)$ and a substitution σ, the operation verifies

$$\left.\begin{array}{l} \sigma \in mgu(t_i, t_j) \\ \bar{t} \in Cc(\delta) \end{array}\right\} \Rightarrow \bar{t}\sigma \in Cc(\delta').$$

The implementation is simple and uses the $\Re$-$\mathtt{UNIF}$ operation (first case) followed by an $\mathtt{FCTA}$ operation.

Implementation 3 $\mathtt{UNIF1}(i,j,\delta) = \delta'$ *where*

$$\begin{array}{lcl} \delta' & = & \mathtt{FCTA}(i,j,\delta_1) \\ frm_1 & = & frm \\ \ell_1 & = & \Re\text{-}\mathtt{UNIF}(\ell,i,j)\ (fi) \\ fi_1 & = & fi \end{array}$$

The consistency proof is straightforward. We provide it to show on a simple example how functions fi can be used in proofs.

Theorem 3 *Operation* $\mathtt{UNIF1}$ *is consistent.*

Proof Assume $\bar{u} \in Cc(\delta)$ and $\sigma \in mgu(u_i, u_j)$. By definition of $Cc(\delta)$, there exists $\bar{t} = (t_1, \ldots, t_p) \in Cc(frm) \cap Cc(l)$ such that $\bar{u} = (t_{fi(1)}, \ldots, t_{fi(q)})$. By definition of $\Re$-$\mathtt{UNIF}(\ell,i,j)$ (fi), $\bar{t}\sigma \in Cc(frm_1) \cap Cc(\ell_1)$. Therefore

$$\begin{array}{lcl} \bar{u}\sigma & = & (t_{fi(1)}\sigma, \ldots, t_{fi(q)}\sigma) \\ & = & (t_{fi_1(1)}\sigma, \ldots, t_{fi_1(q)}\sigma) \\ & \in & Cc(\delta_1). \end{array}$$

Finally, as $u_i\sigma = u_j\sigma$ and $\bar{u}\sigma \in Cc(\delta_1)$, $\bar{u}\sigma \in Cc(\delta')$, by specification of $\mathtt{FCTA}$. $\square$

4.4.5 The Sub-Operation $\mathtt{SPECAT}(i,j,\delta) = \delta'$

The next operation is useful for the unification of two terms t_i, t_j where $frm(i) = undef$ and $frm(j) = f(j_1, \ldots, j_n)$. In fact, such a unification can be achieved in two steps:

1. the unification of t_i and $f(y_1, \ldots, y_n)$ giving σ where $y_1, \ldots, y_n$ are new variables;

2. the unification of $(y_1, \ldots, y_n)\sigma$ and $t_{j_1}, \ldots, t_{j_n}$.

The operation $\mathtt{SPECAT}$ performs the first step. The second step is carried out by the general unification procedure.

Specification 4 *Given $\bar{t} = (t_1, \ldots, t_p)$, a p_tuple of terms, σ a substitution, $y_1, \ldots, y_n$, n distinct variables not occurring in $\bar{t}$, the operation* SPECAT(i, j, δ)= δ' *verifies*

$$\left.\begin{array}{l} \bar{t} \in Cc(\delta) \\ t_i, t_j \textit{ are unifiable} \\ \sigma \in mgu(f(y_1, \ldots, y_n), t_i) \end{array}\right\} \Rightarrow (t_1, \ldots, t_p, y_1, \ldots, y_n)\sigma \;\in\; Cc(\delta').$$

The implementation makes use of a function *reachable* which computes the set of indices reachable from a given index.

Definition 17 *The set of indices reachable from $i \in I_p$ via frm, denoted $reachable(i, frm)$, is defined inductively by*

1. $\{i\}$ *if* $frm(i) = undef$;
2. $\{i\} \bigcup \cup_{j=1}^{n} reachable(i_j, frm)$ *if* $frm(i) = f(i_1, \ldots, i_n)$.

Implementation 4 *The implementation returns $\delta' = \perp$ if $i \in reachable(j, frm)$. Otherwise let $p' = p + n$. The $\Re$ component is obtained by adding to ℓ the correspondence between the term t_i and the set of terms $\{t_{p+1}, \ldots, t_{p+n}\}$, the pattern component is defined by adding the new pattern, and the new function fi': $I_{q+n} \rightarrow I_{p+n}$ is obtained by including the new indices. More precisely,*

$$\begin{array}{lcll} frm' & = & frm \cup \{\langle i, f(p+1, \ldots, p+n)\} \quad (\textit{fi}); & \\ \ell' & = & \Re\text{-}\mathtt{UNIF}(\ell, i, f/n) \quad (\textit{fi}) & \\ \textit{fi}'(k) & = & \textit{fi}(k) & \textit{if } k \leq q \\ \textit{fi}'(k) & = & k - q + p & \textit{if } k > q. \end{array}$$

4.4.6 The Sub-Operation UNIF$(i, j, \delta) = \delta'$

Let us present the main procedure for unification UNIF$(i, j, \delta) = \delta'$ which consists mainly of the three cases mentioned in the overview. Observe that there is no direct reference to the $\Re$-component: the behaviour of the $\Re$-component is completely captured by the operations UNIF1, FCTA and SPECAT defined above.

Informally speaking, procedure UNIF(i, j, δ) unifies subterms i and j in the δ-tuple δ. In the following, we say that $(u_1, \ldots, u_m)$ is a prefix of $(t_1, \ldots, t_n)$ $(m \leq n)$ iff $u_i = t_i$ $(1 \leq i \leq m)$.

Specification 5 *Given $\bar{u} = (u_1, \ldots, u_q)$, a q_tuple of terms, and σ a substitution, the operation verifies*

$$\left.\begin{array}{l} \sigma \in mgu(u_i, u_j) \\ \bar{u} \in Cc(\delta) \end{array}\right\} \Rightarrow \exists \bar{u}' \;\in\; Cc(\delta') : \bar{u}\sigma \text{ is a prefix of } \bar{u}'.$$

Implementation 5 *The implementation of operation* UNIF *is as follows:*

if $i = j$ $(\textit{fi})$ **then**
  $\delta' = \delta$

else if $(frm(i) = undef = frm(j) \quad (fi))$ **then**
$\delta' = \mathtt{UNIF1}(i, j, \delta)$

else if $((frm(i) = undef \quad (fi))\ \&\ \mathtt{SPECAT}(i, j, \delta) = \bot)\ \vee$
$((frm(j) = undef \quad (fi))\ \&\ \mathtt{SPECAT}(j, i, \delta) = \bot)\ \vee$
$((frm(i) = f(i_1, \ldots, i_n)\ \&\ frm(j) = g(j_1, \ldots, j_m) \quad (fi))\ \&\ (f \neq g \vee n \neq m)$ **then**
$\delta' = \bot$

else
$\delta' = \mathtt{FCTA}(i, j, \delta_n)$ **where**

$$\delta_\circ = \begin{cases} \mathtt{SPECAT}(i, j, \delta) & \textbf{if } (frm(i) = undef \quad (fi)) \\ \mathtt{SPECAT}(j, i, \delta) & \textbf{if } (frm(j) = undef \quad (fi)) \\ \delta & \textbf{otherwise} \end{cases}$$

$$(frm_0(i) = f(i_1, \ldots, i_n)\ \ \&\ \ frm_0(j) = f(j_1, \ldots, j_n) \quad (fi_0))$$

$$\delta_k = \mathtt{UNIF}(i_k, j_k, \delta_{k-1}) \ \ (1 \leq k \leq n)\,.$$

The implementation mimics a recursive algorithm for concrete unification as long as at least one of the patterns of u_i and u_j are known. The correctness of this algorithm has been proven in [48] in terms of the specifications of SPECAT, UNIF1 and FCTA which are independent from the $\Re$-domain.

4.4.7 Operation Pat($\Re$)-UNIF(β, x_i, x_j)

We now describe the operation of Pat($\Re$) to unify two terms associated with the variables x_i and x_j in an abstract substitution β. This operation simply unifies the corresponding subterms to obtain a new δ-tuple. The pattern component and the $\Re$-component are inherited directly from the δ-tuple while the same-value component is obtained from the old same-value component and the function of the δ-tuple.

Specification 6 *The operation* Pat($\Re$)-UNIF(β, x_i, x_j) *produces a substitution* β' *such that*

$$\theta \in Cc(\beta)\ \&\ \sigma \in mgu(x_i\theta, x_j\theta) \quad \Rightarrow \quad \theta\sigma \in Cc(\beta').$$

Implementation 6 *The implementation of* Pat($\Re$)-UNIF(β, x_i, x_j) *with* $\beta = (frm, sv, \ell)$ *produces a substitution* $\beta' = (frm', sv', \ell')$ *with*

$$(frm', \ell', fi') = \mathtt{UNIF}(sv(x_i), sv(x_j), (frm, \ell, id)) \ \textit{and}\ sv' = fi' \circ sv$$

where id denotes the identity function.

Clearly, the consistency of the implementation above relies on the correctness of UNIF.

$$\beta = \begin{cases} sv & = & \{x_1 \mapsto 1, x_2 \mapsto 3\} \\ frm & = & \{1 \mapsto g(2,2), 3 \mapsto g(4,5), 5 \mapsto f(6,7)\} \\ l & = & (1 \leftrightarrow 2) \wedge (3 \leftrightarrow 5) \wedge 4 \wedge (5 \leftrightarrow (6 \wedge 7)) \end{cases}$$

$$\delta_0 = (frm_0, l_0, \mathit{fi}_0) = (frm, l, id)$$

1 $\underline{\texttt{CALL}}$ $\texttt{Pat(}\Re\texttt{)-UNIF}(\beta, x_1, x_2)$

2 $\underline{\texttt{CALL}}$ $\texttt{UNIF}(1, 3, \delta_0)$

3 $\underline{\texttt{CALL}}$ $\texttt{UNIF}(2, 4, \delta_0)$

4 $\underline{\texttt{CALL}}$ $\texttt{UNIF1}(2, 4, \delta_0)$

5 $$\delta_{01} = \begin{cases} frm_{01} & = & \{1 \mapsto g(2,2), 3 \mapsto g(4,5), 5 \mapsto f(6,7)\} \\ l_{01} & = & 1 \wedge 2 \wedge (3 \leftrightarrow 5) \wedge 4 \wedge (5 \leftrightarrow (6 \wedge 7)) \\ \mathit{fi}_{01} & = & \mathit{fi}_0 \end{cases}$$

6 $\underline{\texttt{CALL}}$ $\texttt{FCTA}(2, 4, \delta_{01})$

7 $$\delta_1 = \begin{cases} frm_1 & = & \{1 \mapsto g(2,2), 3 \mapsto g(2,4), 4 \mapsto f(5,6)\} \\ l_1 & = & 1 \wedge 2 \wedge (3 \leftrightarrow 4) \wedge (4 \leftrightarrow (5 \wedge 6)) \\ \mathit{fi}_1 & = & \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 4, 6 \mapsto 5, 7 \mapsto 6\} \end{cases}$$

8 $\underline{\texttt{EXIT}}$ $\texttt{FCTA}(2, 4, \delta_{01}) = \delta_1$

9 $\underline{\texttt{EXIT}}$ $\texttt{UNIF1}(2, 4, \delta_0) = \delta_1$

10 $\underline{\texttt{EXIT}}$ $\texttt{UNIF}(2, 4, \delta_0) = \delta_1$

11 $\underline{\texttt{CALL}}$ $\texttt{UNIF}(2, 5, \delta_1)$

12 $\underline{\texttt{CALL}}$ $\texttt{SPECAT}(2, 5, \delta_1)$

13 $$\delta_{10} = \begin{cases} frm_{10} & = & \{1 \mapsto g(2,2), 2 \mapsto f(7,8), 3 \mapsto g(2,4), 4 \mapsto f(5,6)\} \\ l_{10} & = & 1 \wedge 2 \wedge (3 \leftrightarrow 4) \wedge (4 \leftrightarrow (5 \wedge 6)) \wedge 7 \wedge 8 \\ \mathit{fi}_{10} & = & \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 4, 6 \mapsto 5, 7 \mapsto 6, 8 \mapsto 7, 9 \mapsto 8\} \end{cases}$$

14 $\underline{\texttt{EXIT}}$ $\texttt{SPECAT}(2, 5, \delta_1) = \delta_{10}$

15 $\underline{\texttt{CALL}}$ $\texttt{UNIF}(8, 6, \delta_{10})$

16 $\underline{\texttt{CALL}}$ $\texttt{UNIF1}(8, 6, \delta_{10})$

17 $$\delta_{101} = \begin{cases} frm_{101} & = & \{1 \mapsto g(2,2), 2 \mapsto f(7,8), 3 \mapsto g(2,4), 4 \mapsto f(5,6)\} \\ l_{101} & = & 1 \wedge 2 \wedge (3 \leftrightarrow 4) \wedge (4 \leftrightarrow 6) \wedge 5 \wedge 7 \wedge 8 \\ \mathit{fi}_{101} & = & \mathit{fi}_{10} \end{cases}$$

18 $\underline{\texttt{CALL}}$ $\texttt{FCTA}(8, 6, \delta_{101})$

19 $$\delta_{11} = \begin{cases} frm_{11} & = & \{1 \mapsto g(2,2), 2 \mapsto f(5,7), 3 \mapsto g(2,4), 4 \mapsto f(5,6)\} \\ l_{11} & = & 1 \wedge 2 \wedge (3 \leftrightarrow 4) \wedge (4 \leftrightarrow 6) \wedge 5 \wedge 7 \\ \mathit{fi}_{11} & = & \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 4, 6 \mapsto 5, 7 \mapsto 6, 8 \mapsto 5, 9 \mapsto 7\} \end{cases}$$

20 $\underline{\texttt{EXIT}}$ $\texttt{FCTA}(8, 6, \delta_{101}) = \delta_{11}$

21 $\underline{\texttt{EXIT}}$ $\texttt{UNIF1}(8, 6, \delta_{10}) = \delta_{11}$

22 $\underline{\texttt{EXIT}}$ $\texttt{UNIF}(8, 6, \delta_{10}) = \delta_{11}$

Figure 1: Execution of $\texttt{Pat(}\Re\texttt{)-UNIF}$

$\underline{\texttt{CALL}}$ $\texttt{UNIF}(9,7,\delta_{11})$

$\underline{\texttt{CALL}}$ $\texttt{UNIF1}(9,7,\delta_{11})$

$$\delta_{111} = \begin{cases} frm_{111} & = & \{1 \mapsto g(2,2), 2 \mapsto f(5,7), 3 \mapsto g(2,4), 4 \mapsto f(5,6)\} \\ l_{111} & = & 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6 \wedge 7 \\ fi_{111} & = & fi_{11} \end{cases}$$

$\underline{\texttt{CALL}}$ $\texttt{FCTA}(9,7,\delta_{111})$

$$\delta_{12} = \begin{cases} frm_{12} & = & \{1 \mapsto g(2,2), 2 \mapsto f(5,6), 3 \mapsto g(2,4), 4 \mapsto f(5,6)\} \\ l_{12} & = & 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6 \\ fi_{12} & = & \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 4, 6 \mapsto 5, 7 \mapsto 6, 8 \mapsto 5, 9 \mapsto 6\} \end{cases}$$

$\underline{\texttt{EXIT}}$ $\texttt{FCTA}(9,7,\delta_{111}) = \delta_{12}$

$\underline{\texttt{EXIT}}$ $\texttt{UNIF1}(9,7,\delta_{11}) = \delta_{12}$

$\underline{\texttt{EXIT}}$ $\texttt{UNIF}(9,7,\delta_{11}) = \delta_{12}$

$\underline{\texttt{CALL}}$ $\texttt{FCTA}(2,5,\delta_{12})$

$$\delta_{2} = \begin{cases} frm_{2} & = & \{1 \mapsto g(2,2), 2 \mapsto f(4,5), 3 \mapsto g(2,2)\} \\ l_{2} & = & 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \\ fi_{2} & = & \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 2, 6 \mapsto 4, 7 \mapsto 5, 8 \mapsto 4, 9 \mapsto 5\} \end{cases}$$

$\underline{\texttt{EXIT}}$ $\texttt{FCTA}(2,5,\delta_{12}) = \delta_{2}$

$\underline{\texttt{EXIT}}$ $\texttt{UNIF}(2,5,\delta_{1}) = \delta_{2}$

$\underline{\texttt{CALL}}$ $\texttt{FCTA}(1,3,\delta_{2})$

$$\delta_{3} = \begin{cases} frm_{3} & = & \{1 \mapsto g(2,2), 2 \mapsto f(3,4)\} \\ l_{3} & = & 1 \wedge 2 \wedge 3 \wedge 4 \\ fi_{3} & = & \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 1, 4 \mapsto 2, 5 \mapsto 2, 6 \mapsto 3, 7 \mapsto 4, 8 \mapsto 3, 9 \mapsto 4\} \end{cases}$$

$\underline{\texttt{EXIT}}$ $\texttt{FCTA}(1,3,\delta_{2}) = \delta_{3}$

$\underline{\texttt{EXIT}}$ $\texttt{UNIF}(1,3,\delta_{0}) = \delta_{3}$

$$\beta' = (fi_3 \circ sv, frm_3, l_3) = \begin{cases} sv' & = & \{x_1 \mapsto 1, x_2 \mapsto 1\} \\ frm' & = & \{1 \mapsto g(2,2), 2 \mapsto f(3,4)\} \\ f' & = & 1 \wedge 2 \wedge 3 \wedge 4 \end{cases}$$

$\underline{\texttt{EXIT}}$ $\texttt{Pat}(\Re)\texttt{-UNIF}(\beta, x_1, x_2) = \beta'$

Figure 2: Execution of `Pat(ℜ)-UNIF` - cont.

4.4.8 Example

Operation Pat($\Re$)-UNIF is illustrated in Figures 1 and 2 on an abstract substitution β which represents, for instance, the concrete substitution $\{x_1 \mapsto g(y_1, y_1), x_2 \mapsto g(f(a,b), f(y_3, y_4))\}$.

Operation Pat($\Re$)-UNIF(β, x_1, x_2) simply calls the general unification procedure on subterms 1 and 3 associated with variables x_1 and x_2 recursively (line 2). Since both subterms have a well-defined pattern, the unification is called on their arguments, i.e. subterms 2 and 4 (lines 3-10) and subterms 2 and 5 (lines 11-34). The unification is completed by the operation FCTA (line 35) which merges subterms 1 and 3.

The unification of subterms 2 and 4 (lines 3-10) is performed through operation UNIF1 since both of them are leaves. Note the propagation of groundness from subterm 4 to subterm 2 (line 5) and the removal of subterm 4 in the subsequent FCTA operation (line 7).

The unification of subterms 2 and 5 (lines 11-34) is more complex since subterm 2 is a leaf and subterm 5 is not. Operation SPECAT is used (line 12) to make sure that subterm 2 has a well-defined pattern. Note the introduction of subterms 7 and 8 which inherit the groundness of subterm 2. Unification is applied recursively on subterms 8 and 6 (lines 15-22) and subterms 9 and 7 (lines 23-30). In both cases, the basic unification operation UNIF1 is applied.

At the exit of UNIF$(9, 7, \delta_{11})$ only five subterms are left and the subsequent operation FCTA$(1, 3, \delta_2)$ removes one more term by merging subterms 1 and 3 (line 36). The resulting abstract substitution β' (line 39) contains the expected information: x_1 and x_2 are mapped to $g(f(t_1, t_2), f(t_1, t_2))$ where t_1 and t_2 are ground terms.

4.5 Generic Implementation of the Constrained Mapping

We conclude this section by presenting and illustrating a generic implementation of the constrained mapping. This implementation is expressed in terms of simpler operations of the domain Pat($\Re$), each of which can be viewed as a particular case of the constrained mapping. In doing so, our intention is twofold:

1. showing that the constrained mapping is a natural notion that will exist on reasonable domains, since the operations considered are rather elementary;

2. indicating that the designer has two choices in implementing the $\Re$-domain: either implementing directly the constrained mapping or implementing the more elementary operations introduced here; the first option is more efficient but also more demanding for the designer.

The operations are required to be monotone and consistent abstractions of the following concrete operations:

Projection: This operation projects out of term t_j.

$$\mathcal{C}\text{-PROJ}(\Phi, j) = \{< t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_p > \mid < t_1, \ldots, t_p > \in \Phi\}.$$

This operation can be extended easily to sets of indices.

$$\begin{array}{lcl} \mathcal{C}\text{-PROJ}(\Phi, \emptyset) & = & \Phi \\ \mathcal{C}\text{-PROJ}(\Phi, \{j_1, \ldots, j_n\}) & = & \mathcal{C}\text{-PROJ}(\mathcal{C}\text{-PROJ}(\Phi, j_1), \{j_2, \ldots, j_n\}) \text{ where } j_1 = max(\{j_1, \ldots, j_n\}). \end{array}$$

Renaming: This operation permutes some of the elements. Let $r : I_p \to I_p$ be a permutation of indices.

$$\mathcal{C}\text{-}\mathtt{REN}(\Phi, r) \quad = \quad \{< t_{r(1)}, \ldots, t_{r(p)} > | \quad < t_1, \ldots, t_p > \in \Phi\}.$$

Duplication: This operation duplicates an element.

$$\mathcal{C}\text{-}\mathtt{DUP}(\Phi, i) \quad = \quad \{ < t_1, \ldots, t_i, \ldots, t_p, t_i > | \quad < t_1, \ldots, t_i, \ldots, t_p > \in \Phi \}.$$

Given a sequence of indices $\langle i_1, \ldots, i_n \rangle$, we define

$$\begin{array}{lcll} \mathcal{C}\text{-}\mathtt{DUP}(\Phi, \langle i_1, \ldots, i_n \rangle) & = & \mathcal{C}\text{-}\mathtt{DUP}(\mathcal{C}\text{-}\mathtt{DUP}(\Phi, i_1), \langle i_2, \ldots, i_n \rangle) & (n \geq 1) \\ \mathcal{C}\text{-}\mathtt{DUP}(\Phi, \langle\rangle) & = & \Phi & \end{array}$$

Example 6 *We illustrate how these operations can be implemented for* `Pos`*. For this purpose, we define the denormalization* `denorm` $[i_1, \ldots, i_n]\ f$ *of a formula* f *wrt* $[i_1, \ldots, i_n]$ *as the boolean function obtained by replacing simultaneously* $1, \ldots, n$ *by* $i_1, \ldots, i_n$ *in* f*. We also define the valuation* $f|_{i=b}$ *of a function* f *wrt index* i *and a truth value* b *as the function obtained by replacing* i *by* b *in* f*. The new operations on* `Pos` *can then be defined as follows:*

$\mathtt{Pos\text{-}PROJ}(f, j) = \mathtt{denorm}\ [1, \ldots, j-1, j, j, \ldots, p-1]\ (f|_{j=\mathtt{true}} \vee f|_{j=\mathtt{false}})$.

$\mathtt{Pos\text{-}REN}(f, r) = \mathtt{denorm}\ [r(1), \ldots, r(p)]\ f$.

$\mathtt{Pos\text{-}DUP}(f, i) = f \wedge (i \leftrightarrow p+1)$.

Note that $p+1$ *is the new index is the last operation.*

We are now in position to define the constrained mapping in a generic way.

Implementation 7 *The constrained mapping* $tr^{\#}$ *of* $tr : I_{p_2} \rightarrow I_{p_1}$ *can be defined as follows. Let*

$p_3 = \#tr(I_{p_2})$ *where* $\#A$ *denotes the cardinality of a set* A*;*
$tr(I_{p_2}) = \{i_1, \ldots, i_{p_3}\}$ *such that* $i_1 < \ldots < i_{p_3}$*;*
$tr_1 : I_{p_2} \rightarrow I_{p_2}$ *such that* $\left\{ \begin{array}{ll} 1) & tr_1 \textit{ is a permutation;} \\ 2) & tr(tr_1(j)) = i_j \textit{ for } j \in I_{p_3}; \end{array} \right.$
$tr_2 : I_{p_1} \rightarrow I_{p_3}$ *such that* $tr_2(i_j) = j$ *for* $j \in I_{p_3}$

in

$$\begin{array}{lcl} \ell_1 & = & \Re\text{-}\mathtt{PROJ}(\ell, I_{p_1} \setminus \{i_1, \ldots, i_{p_3}\}) \\ \ell_2 & = & \Re\text{-}\mathtt{DUP}(\ell_1, \langle tr_2(tr(tr_1(p_3+1))), \ldots, tr_2(tr(tr_1(p_2))) \rangle) \\ tr^{\#}(\ell) & = & \Re\text{-}\mathtt{REN}(\ell_2, tr_1^{-1}). \end{array}$$

As mentioned previously, the key idea is to project irrelevant terms and to introduce new terms and equality constraints to obtain the new domain. Note that tr_1 in the implementation can be defined as follows.

$$\begin{array}{lcl} V & = & \{j \ | \ j \in I_{p_2} \ \& \ \forall k \in I_{p_2} : tr(k) = tr(j) \Rightarrow j \leq k\} \\ tr_1(j) & = & min\{k \mid tr(k) = i_j\} \text{ if } j \leq p_3 \\ & = & k_{j-p_3} \text{ if } j > p_3 \text{ where } k_1 < \ldots < k_{p_2 - p_3} \ \& \ V \cup \{k_1, \ldots, k_{p_2-p_3}\} = I_{p_2}. \end{array}$$

Theorem 4 *Implementation 7 of the constrained mapping is consistent.*

Proof:

$$\begin{array}{ll}
 & \langle t_1, \ldots, t_{p_1} \rangle \in Cc(l) \\
\Rightarrow & \langle t_{i_1}, \ldots, t_{i_{p_3}} \rangle \in Cc(l_1) \\
\Rightarrow & \langle t_{tr(tr_1(1))}, \ldots, t_{tr(tr_1(p_3))} \rangle \in Cc(l_1) \\
\Rightarrow & \langle t_{tr(tr_1(1))}, \ldots, t_{tr(tr_1(p_3))}, t_{tr(tr_1(tr_2(tr(tr_1(p_3+1)))))}, \ldots, t_{tr(tr_1(tr_2(tr(tr_1(p_2)))))} \rangle \in Cc(l_2) \\
\Rightarrow & \langle t_{tr(tr_1(1))}, \ldots, t_{tr(tr_1(p_3))}, t_{tr(tr_1(p_3+1))}, \ldots, t_{tr(tr_1(p_2))} \rangle \in Cc(l_2) \\
\Rightarrow & \langle t_{tr(tr_1(tr_1^{-1}(1)))}, \ldots, t_{tr(tr_1(tr_1^{-1}(p_2)))} \rangle \in Cc(tr^{\#}(l)) \\
\Rightarrow & \langle t_{tr(1)}, \ldots, t_{tr(p_2)} \rangle \in Cc(tr^{\#}(l))
\end{array}$$

■

Example 7 *Consider again Example 5 and let us compute $tr_1^{\#}$ and $tr_2^{\#}$ according to Implementation 7.*

a) $tr_1 : I_6 \to I_7; \qquad tr_1 = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 6, 5 \mapsto 7, 6 \mapsto 7\}$

$$tr_{1_1} : I_6 \to I_6; \qquad tr_{1_1} = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 6 \mapsto 6\}$$
$$tr_{1_2} : I_7 \to I_5; \qquad tr_{1_2} = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 6 \mapsto 4, 7 \mapsto 5\}$$

$$l_1' = \texttt{Pos-PROJ}(l_1, \{4,5\}) = (1 \leftrightarrow (2 \land 3)) \land (3 \leftrightarrow 5) \land (4 \leftrightarrow 5)$$
$$l_1'' = \texttt{Pos-DUP}(l_1', tr_{1_2}(tr_1(tr_{1_1}(6))) = \texttt{Pos-DUP}(l_1', 5) = l_1' \land (5 \leftrightarrow 6)$$
$$tr_1^{\#}(l_1) = \texttt{Pos-REN}(l_1'', tr_{1_1}^{-1}) = (1 \leftrightarrow (2 \land 3)) \land (3 \leftrightarrow 5) \land (4 \leftrightarrow 5) \land (5 \leftrightarrow 6).$$

b) $tr_2 : I_6 \to I_5; \qquad tr_2 = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 6 \mapsto 3\}.$

$$tr_{2_1} : I_6 \to I_6; \qquad tr_{2_1} = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 6 \mapsto 6\}$$
$$tr_{2_2} : I_5 \to I_5; \qquad tr_{2_2} = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5\}$$

$$l_2' = \texttt{Pos-PROJ}(l_2, \emptyset) = l_2$$
$$l_2'' = \texttt{Pos-DUP}(l_2', tr_{2_2}(tr_2(tr_{2_1}(6))) = \texttt{Pos-DUP}(l_2', 3) = l_2' \land (3 \leftrightarrow 6)$$
$$tr_2^{\#}(l_2) = \texttt{Pos-REN}(l_2'', tr_{2_1}^{-1}) = (1 \leftrightarrow (2 \land 3)) \land (4 \leftrightarrow (5 \land 3)) \land (3 \leftrightarrow 6)$$

4.6 Applications

The simplest applications of `Pat(`$\Re$`)` amount to upgrading a single domain. Examples are the domain `Pat(Pos)` for groundness analysis and the domain `Pat(Type)`, upgrading the rigid type graph of Bruynooghe and Janssens [5] for type analysis. `Pat(Pos)` [58] produces perfectly accurate results for our suite of benchmarks[15], improving on the domain `Pos` for programs manipulating difference lists. Note that it is clear that an example losing accuracy can be constructed. `Pat(Type)` is a very complex domain [59] inferring automatically recursive and disjunctive types.

Another applications of `Pat(`$\Re$`)` consists in having $\Re$ as an open product, combining the two contributions of this work. The domains `Pat(OPos` $\otimes$ `OPS)` and `Pat(OPos` $\otimes$ `OPS` $\otimes$ `OMode)`,

[15] The benchmarks are available by anonymous ftp from Brown University and are used by several research groups.

where `OMode` is a mode domain [48] assigning to each subterms a mode from { `var, ground, ngv` *(neither ground nor variable)*, `novar, noground, gv` *(either ground or variable)*, `any` } , have been built along these lines.

Finally, more advanced domains can be built by defining $\Re$ as an open domain which can receive structural information from the pattern component. Although most domains will not need this information, a mode domain maintaining information on all subterms may benefit from this interaction. For $\Re$ to be an open domain in this case, it is necessary to generalize slightly the open product such that its operations can be open operations as well.

The domain `OPat(OPS` $\otimes$ `OMode)`, used to quantify the loss of efficiency of our approach, was defined using this approach.

Note also that for most of our benchmarks, the computation times are below 10 seconds, even for complex domains such as `Pat(OPos` $\otimes$ `OPS` $\otimes$ `OMode)` and `Pat(Type)`.

5 Experimental Evaluation

In this section, we briefly describe experimental results to indicate the practical interest of our approach. We describe the reduction in development effort, discuss respectively open operations and refinements and assess the overhead of our approach. The results were obtained with `GAIA` [35], all domains being implemented in C and the system being run on a Sun SS30/10.

The Programs Tested The programs we use are hopefully representative of "pure" logic programs (i.e. without the use of dynamic predicates such as `assert` and `retract`). They are taken from a number of authors and used for various purposes from compiler writing to equation-solvers, combinatorial problems, and theorem-proving. Hence they should be representative of a large class of programs. In order to accommodate the many built-ins provided in Prolog implementations and not supported in our current implementation, some programs have been extended with some clauses achieving the effect of the built-ins. Examples are the predicates to achieve input/output, meta-predicates such as `setof`, `bagof`, `arg`, and `functor`. The clauses containing `assert` and `retract` have been dropped in the one program containing them (i.e. Syntax error handling in the reader program).

The program `kalah` is a program which plays the game of kalah. It is taken from [53] and implements an alpha-beta search procedure. The program `press` is an equation-solver program taken from [53] as well. We use two versions of this program, `press1` and `press2`, the difference being that `press2` has a procedure call repeated in the body of a procedure. The program `cs` is a cutting-stock program taken from [55]. It is a program used to generate a number of configurations representing various ways of cutting a wood board into small shelves. The program uses, in various ways, the nondeterminism of Prolog. We use two versions of the program; one of them (i.e. `cs1`) assumes that the data are ground while the other one (i.e. `cs`) assumes that the data are ground lists. The program `disj` is taken from [23] and is the generate and test equivalent of a constraint program used to solve a disjunctive scheduling problem. This is also a program using the nondeterminism of Prolog. Once again, we use two versions of the program with the same distinction as for the cutting stock example. The program `read` is the tokeniser and reader written by R. O'keefe and D.H.D. Warren for Prolog. It is mainly a deterministic program, with mutually recursive procedures. The program `pg` is a program written by W. Older to solve a

	N	Pos	Pat(Pos)
Press1	143	15	99
Press2	143	15	99

Table 1: Importance of Structural Information: Accuracy of the Analysis on Inputs. `N` is the number of predicate arguments and `Pos` and `Pat(Pos)` are the numbers of ground arguments inferred by the domains.

specific mathematical problem. The program `gabriel` is the `Browse` program taken from Gabriel benchmarks. The program `plan` is a planning program taken from Sterling & Shapiro. The program `queens` is a simple program to solve the n-queens problem. `peep` is a program written by S.Debray to carry out the *peephole* optimization in the SB-Prolog compiler. It is a deterministic program. We also use the traditional concatenation and quicksort programs, say `append` (with input modes `(var,var,ground)`) and `qsort` (difference lists).

The Domains Tested The domains used in our experimental results are the domains `OPos` and `PPS` presented earlier as well a mode domain `OMode` inspired by [35]. This last domain assigns a mode from { `var, ground, ngv, novar, noground, gv, any` } to each index.

On the Development Effort We first give some ideas about the effort necessary to produce the sophisticated domain `OPAT(OPos`$\otimes$`OMode`$\otimes$`OPS)`. The overall implementation of the system is 17,712 lines of C, split in 15,759 lines in `.c` files (programs) and `1,953` lines in `.h` files (data structure definitions). The mode component requires 822 lines (785 + 37), the sharing component requires 800 lines (761 + 39), and the `Pos` component requires `1791` lines (1766 + 25). For this application, only 19% of the overall code needs to be supplied. Domain `OPAT(OMode`$\otimes$`OPS)` needs only to produce about 10% of the overall code. Its domain part (1622 lines) produces a reduction of about 40% over the direct implementation (i.e. the domain `Pattern` [35][35][16]) which requires 2657 lines (2463 + 194). As should be clear, our approach reduces the development effort substantially. Note also that the above figures do not account for the support in the design process, which allows designers to concentrate on one domain at a time and to be liberated from structural information.

On the Importance of Structural Information Structural information may improve accuracy substantially and strongly reduces the impact of the syntax of the programs on the accuracy of the analysis. The gain produced by structural information has also been measured by several authors (e.g. [28, 38]). We will not repeat those results here. Rather we show that even for very accurate domains such as `Pos`, preserving structural information improves accuracy on practical programs.

Tables 1 and 2 compare `Pos` and `Pat(Pos)` for the input and output patterns and only report the programs for which there are some differences. The results indicate that `Pat(Pos)` improves on `Pos` on the `press` programs as far as inputs are concerned and on the `press` programs and `read` for the outputs. The improvement comes from the better handling of difference-lists provided by `Pat(Pos)`. Note also that the increase in precision is substantial for the `press` program.

[16] We thus have two implementations of the domain `Pattern`: a direct one and one built using the techniques described in this paper. These two versions will be compared later with respect to efficiency.

	N	Pos	Pat(Pos)
Press1	143	39	140
Press2	143	39	140
Read	122	70	74

Table 2: Importance of Structural Information: Accuracy of the Analysis on Output. `N` is the number of predicate arguments and `Pos` and `Pat(Pos)` are the numbers of ground arguments inferred by the domains.

	Pos			Pat(Pos)			Pat(Pos)/Pos		
	Time	GI	CI	Time	GI	CI	Time	GI	CI
cs	1.34	50	94	20.95	84	166	15.63	1.68	1.77
d isj	1.01	45	88	9.59	68	134	9.50	1.51	1.52
gabriel	0.47	47	114	11.98	62	141	25.49	1.32	1.24
kalah	0.93	65	129	22.52	117	236	24.22	1.80	1.83
peep	1.16	36	249	15.98	76	410	13.78	2.11	1.65
pg	0.16	16	31	2.42	36	76	15.13	2.25	2.45
plan	0.12	19	41	2.50	31	67	20.83	1.63	1.63
press1	5.96	287	866	34.25	190	631	5.75	0.66	0.22
press2	6.03	287	878	34.85	192	655	5.78	0.67	0.22
qsort	0.05	7	15	0.31	10	22	6.20	1.43	1.47
queens	0.04	9	17	0.32	15	29	8.00	1.67	1.71
read	1.66	76	311	182.07	178	804	109.68	2.34	0.57
Mean							21.66	1.59	1.36

Table 3: The Domain `Pat(Pos)`: Efficiency Results. `Time` is the computation time in seconds, `GI` is the number of procedure iterations, `CI` is the number of clauses iterations.

Program	Time-on	Iter-on	Time-st	Iter-st	Time-on/Time-st	Iter-on/Iter-st
CS	39.12	99	20.95	84	1.87	1.18
Disj	53.14	74	9.59	68	5.54	1.09
Kalah	34.80	130	22.52	117	1.55	1.11
Peep	36.93	80	15.98	76	2.31	1.05
PG	2.66	37	2.42	36	1.10	1.03
Plan	3.27	40	2.50	31	1.31	1.29
Press1	33.85	190	34.26	190	0.99	1.00
Press2	34.35	192	34.85	192	0.99	1.00
QSort	0.43	11.00	0.31	10.00	1.39	1.10
Queens	0.65	16.00	0.32	15.00	2.03	1.07
Read	182.07	179.00	182.07	178.00	1.00	1.01
Mean	38.30	95.27	29.62	90.64	1.82	1.08

Table 4: On-line Analysis: Efficiency Results of `Pat(Pos)`. `Time` is the computation time in seconds, while `Iter` is the number of procedure iterations

Table 3 depicts the efficiency results of `Pat(Pos)` and compares them to `Pos`. The times of the analyses with `Pat(Pos)` are reasonable, yet they are substantially slower (about 22 times) than `Pos`. This indicates clearly a tradeoff between efficiency and accuracy in this case.

For completeness, we also consider the use of `Pat(Pos)` for online analysis (see [22] for a definition of on-line analysis and a comparison with usual *global* analysis approaches) where a general analysis of some components is performed once and specialized for the input patterns encountered during subsequent analysis.[17] `Pos` and `Pat(Pos)` are potentially interesting domains for on-line analysis since it is possible to obtain a specialized output pattern by unifying the input pattern and the general output pattern, as `Pos` is condensing [44]. For instance, in `Pos`, $\mathtt{append(x_1,x_2,x_3)}$ returns $x_3 \leftrightarrow x_2 \wedge x_1$, and $\mathtt{qsort(x_1,x_2)}$ returns $x_1 \leftrightarrow x_2$ which can both be specialized optimally. To carry out the experiments, all programs have been run without any assumption on the input patterns (and/or the database) and have been specialized afterwards with the input patterns.

Table 4 gives the efficiency results which compare the online analysis with the traditional analysis. As the result indicates, the online analysis is really practical and is about 1.8 slower than the traditional analysis.

Open Product versus Pseudo-Reduced Product We now investigate the importance of open operations to find out whether refinement operations can recover the loss of information coming from a direct product. The results illustrate directly the benefit of the open product over the pseudo-reduced product operations. We use the domain `OPAT(OMode`$\otimes$`OPS)` for the experimental results in its standard version (denoted by `S`) and in a modified version (denoted `NQ`) where `OMode` and `OPS` can only interact through the refinement operations and implement the pseudo-reduced product. The accuracy results depicted in Tables 5 and 6 demonstrate the importance of open operations. As far as input patterns are concerned, `NQ` loses in the average about 26% accuracy for modes, 81% for freeness (i.e. determining if a predicate argument is an unbound variable or is bound to a variable), 0.42% for groundness (i.e. determining if a predicate argument contains no variable), and has 105% sharing with respect to `S`. As far as output patterns are considered, `NQ` exhibits substantially more sharing than `S` (e.g. up to 50 times more sharing on some of the larger

[17]On-line analysis is also called *goal independent* analysis by some researchers in the logic programming community.

		modes		freeness			groundness			sharing		
	N	loss	%loss	S	NQ	%(S-NQ)/S	S	NQ	%(S-NQ)/S	S	NQ	%NQ/S
cs	94	30	31.91	37	7	81.08	56	56	0.00	38	40	105.26
disj	60	17	28.33	21	5	76.19	38	38	0.00	22	22	100.00
gabriel	59	16	27.12	18	2	88.89	18	18	0.00	44	52	118.18
kalah	123	36	29.27	39	8	79.49	79	79	0.00	44	44	100.00
peep	63	13	20.63	15	2	86.67	39	39	0.00	24	24	100.00
pg	31	7	22.58	6	1	83.33	20	20	0.00	11	11	100.00
plan	32	7	21.88	6	1	83.33	20	19	5.00	13	14	107.69
press1	143	30	20.98	36	8	77.78	15	15	0.00	169	181	107.10
press2	143	31	21.68	36	8	77.78	99	99	0.00	44	44	100.00
qsort	9	2	22.22	3	1	66.67	4	4	0.00	5	5	100.00
queens	11	3	27.27	4	1	75.00	7	7	0.00	4	4	100.00
read	122	4	36.07	46	3	93.48	34	34	0.00	98	118	120.41
Mean			25.83			80.81			0.42			104.89

Table 5: The Importance of Open Operations: Input Results. `N` is the number of arguments. `loss` is the number of arguments losing precision for modes. `S` is used for the standard version with open operations, `NQ` is used for the version where the domains interact only on refinements. The number of free terms, ground terms and pair sharing between terms are computed.

programs). Although they are appropriate to adjust groundness information, refinement operations lose much precision for other measures such as freeness, input modes, and sharing. In these cases, refinements cannot recover the information lost during the operations. It is worth mentioning however that the pseudo-reduced product is appropriate when the goal is to reuse existing domains without modifying the implementation of the operations. The efficiency results depicted in Table 7 show that `S` is slightly more efficient than `NQ` in the average, demonstrating that open operations are particularly appropriate. In the average, `NQ` is 1.05 slower than `S`. Note that `S` is about twice faster than `NQ` on one of the benchmark programs (i.e. `read`).

On the Importance of Refinements We now investigate the importance of refinements in conjunction with open operations to find out whether open operations are sophisticated enough to eliminate the need for refinements. We use the domains `OPAT(OMode⊗OPS)` and `PAT(OPos⊗OPS)` for the experimental results in their standard version (denoted by `S`) and in their modified version (denoted `NR`) where no refinement operations are used. The experimental results show no difference between the versions on the two domains considered. This result is easily explained in the case of `OPAT(OMode⊗OPS)`, since the local nature of the domains makes sure that the initial object together with the operation arguments contain enough information to avoid the need for refinements. In the case of `PAT(OPos⊗OPS)`, no difference is observable at the level of the inputs/outputs but some differences occur during the fixpoint computation. This is due to the global nature of `Pos` which propagates groundness beyond the operation arguments. This indicates that refinements can be useful even in conjunction with open operations, although our experimental results tend to suggest that the improvements will be much less dramatic. Note also that refinements can be useful when the definition of the operations are not optimal in precision. In a previous non-optimal version of

		sharing		
	N	S	NQ	%NQ/S
cs	94	0	116	∞
disj	60	0	73	∞
gabriel	59	55	58	105.45
kalah	123	2	114	5700.00
peep	63	11	103	936.36
pg	31	0	40	∞
plan	32	1	53	5300.00
press1	143	179	208	116.20
press2	143	3	208	6933.33
qsort	9	4	11	275.00
queens	11	0	13	∞
read	122	86	175	203.49
Mean				∞

Table 6: The Importance of Open Operations: Output Results. `N` is the number of arguments. `S` is used for the standard version with open operations. `NQ` is used for the version where the domains interact only on refinements.

	S (Standard Version)			NQ (No Query Version)			NQ/S		
	Time	GI	CI	Time	GI	CI	Time	GI	CI
cs	4.75	85	168	4.63	84	164	0.97	0.99	0.98
disj	2.45	68	134	2.56	69	136	1.04	1.01	1.01
gabriel	1.64	81	190	1.39	71	163	0.85	0.88	0.86
kalah	4.41	117	236	4.59	119	239	1.04	1.02	1.01
peep	5.03	94	530	3.68	71	360	0.73	0.76	0.68
pg	0.66	38	80	0.62	36	76	0.94	0.95	0.95
plan	0.57	36	78	0.58	33	72	1.02	0.92	0.92
press1	24.06	554	1841	37.46	544	1909	1.56	0.98	1.04
press2	6.90	212	707	6.92	197	679	1.00	0.93	0.96
qsort	0.16	13	28	0.16	11	24	1.00	0.85	0.86
queens	0.15	15	29	0.08	16	31	0.53	1.07	1.07
read	10.95	209	938	21.10	290	1331	1.93	1.39	1.42
Mean							1.05	1.00	1.01

Table 7: The Importance of Open Operations: Efficiency Results. `N` is the number of arguments. `S` is used for the standard version with open operations. `NQ` is used for the versions where the domains interact only on refinements. `Time` is the computation time in seconds, `GI` is the number of procedure iterations, `CI` is the number of clauses iterations.

	S(tandard Version)	D(irect Implementation)	D/S
cs	4.75	2.13	0.45
disj	2.45	1.15	0.47
gabriel	1.64	0.73	0.45
kalah	4.41	1.99	0.45
peep	5.03	2.33	0.46
pg	0.66	0.29	0.44
plan	0.57	0.20	0.35
press1	24.06	9.07	0.38
press2	6.90	2.90	0.42
qsort	0.16	0.06	0.38
queens	0.15	0.06	0.40
read	10.95	5.41	0.50
Mean			0.43

Table 8: The Overhead of The Approach: Efficiency Results

the sharing component, refinements produce a substantial improvement in precision (`NR` produced 230% of the sharing of `S`) and a slight improvement in efficiency. Moreover, they are especially appropriate when the goal is to reuse existing domains without modifying the operations.

On the Overhead of the Approach We turn to the overhead of our approach in `OPAT(OMode`$\otimes$`OPS)` compared to a direct implementation of our pattern domain [35]. Our approach introduces mainly three forms of overheads: (1) *global operations:* the generic pattern domain has provisions to accommodate global information on subterms which complicates the operations when only local information is used as in `OPAT(OMode`$\otimes$`OPS)`; (2) *memory management:* the approach allocates and deallocates memory with a much smaller granularity because the domains are disconnected; (3) *queries:* the query mechanism introduces an additional layer necessary to combine the domain. The results depicted in Table 8 indicate that the direct implementation requires about 43% of the time of standard version. This is an acceptable overhead given the significant reduction in development time offered by the approach. However, the overhead should be interpreted with care, since the implementation has not be tuned with the same care as the direct implementation. In particular, the overhead can be significantly reduced by improving memory management, caching queries whenever appropriate, and specializing the implementation when the full generality is not needed. This is obviously an important topic for further research.

6 Conclusion

The purpose of this paper was to tackle one of the most important open problems in the design of static analysis of logic programs: the building of abstract domains. This problem is important, since logic program analyses are in general quite sophisticated because of the need to integrate various interdependent analyses and to maintain structural information.

The paper introduced two new ideas: the notion of open product and a generic pattern domain. The open product enables the combination of domains where the components interact through the notions of queries and open operations. It provides a rich framework to build complex combinations of domains. The generic pattern domain upgrades automatically a domain with structural information providing an (often substantial) increase in accuracy at no additional cost in design and implementation. Both contributions have been validated theoretically and experimentally and the experimental results showed the practical benefits of our approach.

Future work on the theory will focus on generalizing the notion of open product in several directions. A promising line of research amounts of viewing all operations as coroutines communicating information whenever appropriate. This may allow to view `Pat(`$\Re$`)` as a product although the theoretical and practical consequences of this view are still to be explored. On the practical side, fine-tuning the implementation and a better environment for designers are the first priorities.

Acknowledgements

Agostino Cortesi is partly supported by MURST project 9701248444. Pascal Van Hentenryck is partly supported by the National Science Foundation under grant number CCR-9108032, the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225, and by a National Young Investigator Award.

References

[1] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol 15(1), pp. 133–181, 1993.

[2] P. Bigot, S. Debray, and K. Marriott. Understanding Finiteness Analysis Using Abstract Interpretation. In *Proceedings of the International Joint Conference and Symposium on Logic Programming (JICSLP-92)*, Washington, DC, November 1992.

[3] A. Bossi, M. Gabbrielli, G Levi, and M-C. Meo. Contribution to the Semantics of Open Logic Programs. In *Proc. of Int. Conf. on Fifth Generation Computer Systems*, Tokyo, June 1992.

[4] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[5] M. Bruynooghe, Janssens, G. Propagation: A New Operation in a Framework for Abstract Interpretation of Logic Programs. In *Meta-Programming in Logic, 3rd International Workshop.* Lecture Notes in Computer Science, Vol. 649, pp. 294-307, 1992.

[6] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[7] M. Codish, and D. Dams, and E. Yardeni. Derivation and safety of an Abstract Unification Algorithm for Groundness and Sharing Analysis. In K. Furukawa, editor, *Proc. Int. Conference of Logic Programming (ICLP'91),* Paris, June 1991, MIT Press, 1991.

[8] M. Codish, A. Mulkers, M. Bruynooghe, M. Garcia de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM93)*, Copenhagen, Denmark, June 1993.

[9] M. Codish, A. Mulkers, M. Bruynooghe, M. Garcia de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems* Vol. 17(1), pp. 28–44, 1995.

[10] P. Codognet and G. Filé. Computations, Abstractions and Constraints in Logic Programs. In *Proceedings of the Fourth International Conference on Programming Languages (ICCL'92)*, Oakland, CA, April 1992.

[11] A. Cortesi and G. Filé. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness, and Compoundness Analysis. In *Proc. ACM-PEPM'91, P. Hudak and N. Jones (eds.), SIGPLAN NOTICES vol.26, n.11, 1991*, 1991.

[12] A. Cortesi, G. Filé, and W. Winsborough. Prop revisited: Propositional formulas as abstract domain for groundness analysis. In *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, pages 322–327, 1991.

[13] A. Cortesi, G. Filé, and W. Winsborough. Comparison of Abstract Interpretations. In *Proc. 19th International Colloquium on Automata, Languages and Programming (ICALP'92)*, 1992.

[14] A. Cortesi, and G. Filé. Comparison and Design of Abstract Domains for Sharing Analysis. In D. Saccà, editor, *Proc. 8th Italian Conference on Logic Programming (GULP'93)*, Gizzeria, Italy, 1991.

[15] A. Cortesi, G. Filé, and W. Winsborough. Optimal Groundness Analysis Using Propositional Logic *The Journal of Logic Programming*, vol. 27(2), pp.137-167, 1996.

[16] P Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, CA, 1977.

[17] P Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Conf. Record of Sixth ACM Symposium on Programming Languages (POPL'79)*, pages 269–282, San Antonio, Tx, 1979.

[18] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2-3), 1992.

[19] S. Debray. On the complexity of dataflow analysis of logic programs. In *Proc. 19th ICALP*, Vienna, Austria, July 1992.

[20] S. Debray. Efficient Dataflow Analysis of Logic Programs. *Journal of ACM* 39(4): 949-984 (1992)

[21] S. Debray and P. Mishra. Denotational and operational semantics for prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.

[22] A. Deutsch. A Storeless Model of Aliasing and its Abstraction Using Finite Representations of Right-Regular Equivalence Relations. In *Fourth IEEE International Conference on Computer Languages (ICCL'92)*, San Francisco, CA, April 1992.

[23] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[24] Englebert, V., Le Charlier, B., Roland D., and Van Hentenryck, P. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and their Experimental Evaluation. Software Practice and Experience, 23(4): 419-459, 1993.

[25] Granger, P. Improving the Results of Static Analyses Programs by Local Decreasing Iteration. In R. K. Shyamasundar (Ed.): *Foundations of Software Technology and Theoretical Computer Science* Lecture Notes in Computer Science, Vol. 652, pp 68-79, 1992.

[26] N. Heintze and J. Jaffar. An Engine for Logic Program Analysis. In *IEEE 7th Annual Symposium on Logic in Computer Science*, 1992.

[27] M. Hermenegildo and K. Muthukumar. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[28] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, 1992.

[29] Horiuchi,K., and Kanamori, T. Polymorphic Type Inference in Prolog by Abstract Interpretation. In *Logic Programming '87, Proceedings of the 6th Conference* Lecture Notes in Computer Science, Vol. 315, pp.195-214, 1987.

[30] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

[31] N.D. Jones and H. Sondergaard. *A Semantics-Based Framework for the Abstract Interpretation of Prolog*, pages 123–142. Ellis Horwood, 1987.

[32] Kifer,M., and Lozinskii,E.L., A framework for an efficient implementation of deductive databases. In *Proc. of the 6-th Advanced Database Symposium*, 1986.

[33] Kifer,M., and Lozinskii,E.L., SYGRAF: Implementing Logic Programs in a Database Style *IEEE Trans. on Software Engineering*, 1988, pp. 922–935.

[34] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and Its Complexity Analysis (Extended Abstract). In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[35] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems* 1993. An extended abstract appeared in the Proceedings of Fourth IEEE International Conference on Computer Languages (ICCL'92), San Francisco, CA, April 1992.

[36] C. Leclre, and B. Le Charlier. Two Dual Abstract Operations to Duplicate, Eliminate, Rename, Introduce, and Equalize Place-Holders Occurring inside Abstract Descriptions. Technical Report RR-98-009, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, June 1998.

[37] B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, CS Department, Brown University, 1992.

[38] B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. In *Proceedings of the International Joint Conference and Symposium on Logic Programming (JICSLP-92)*, Washington, DC, November 1992.

[39] B. Le Charlier and P. Van Hentenryck. Groundness Analysis for Prolog: Implementation and Evaluation of the Domain `Prop`. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM93)*, Copenhagen, Denmark, June 1993.

[40] B. Le Charlier, C. Leclere, S. Rossi, A. Cortesi, Automated Verification of Prolog Programs, *The Journal of Logic Programming,* Special Issue on Synthesis, Transformation and Analysis of Logic Programs. Vol.39(1-3), pp. 3–42 (1999).

[41] K. Marriott and H. Sondergaard. Bottom-up Abstract Interpretation of Logic Programs. In *Proc. Fifth International Conference on Logic Programming*, pages 733–748, Seattle, WA, August 1988.

[42] K. Marriott and H. Sondergaard. Notes for a Tutorial on Abstract Interpretation of Logic Programs. North American Conference on Logic Programming, Cleveland, Ohio, 1989.

[43] K. Marriott and H. Sondergaard. Semantics-based Dataflow Analysis of Logic Programs. In *Information Processing-89*, pages 601–606, San Francisco, CA, 1989.

[44] Marriott, K. and Søndergaard, H., "Precise and Efficient Groundness Analysis for Logic Programs." *ACM LOPLAS* Vol. 2, pp.81–196, 1993.

[45] C. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. Technical Report DAI Report 163, Department of Artificial Intelligence, University of Edinburgh, 1981.

[46] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *Seventh International Conference on Logic Programming (ICLP-90)*, Jerusalem, Israel, June 1990.

[47] Mycroft, A. Completeness and predicate-based abstract interpretation. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM93)*, Copenhagen, Denmark, June 1993.

[48] K. Musumbu. *Interpretation Abstraite de Programmes Prolog.* PhD thesis, University of Namur (Belgium), September 1990.

[49] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information Through Abstract Interpretation. In *Proceedings of the North American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

[50] Nielson, F. Tensor products generalize the relational data flow analysis method. In *Proc 4th Hungarian Comp. Sci. Conf.*, pp. 211-225, 1985.

[51] U. Nilsson. Systematic Semantic Approximations of Logic Programs. In *Proceedings of PLILP 90*, pages 293–306, Linkoping, Sweeden, August 1990.

[52] H. Sondergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In *Proc. of ESOP'86*, pages 327–338, Sarrbruecken (FRG), 1986.

[53] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, Ma, 1986.

[54] Taylor, A. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In David H. D. Warren, Peter Szeredi (Eds.): *Logic Programming, Proceedings of the Seventh International Conference*, MIT Press, pp 174-185, 1990.

[55] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[56] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. Abstract Interpretation of Prolog Based on OLDT-Resolution. Technical Report No. CS-93-05, CS Department, Brown University, 1993.

[57] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The impact of Granularity in Abstract Interpretation of Prolog. In Cousot, P., and al. (Eds.): *Static Analysis, Third International Workshop* Lecture Notes in Computer Science, Vol. 724, Springer 1993.

[58] Van Hentenryck P., Cortesi, A. and Le Charlier, B., Evaluation of the domain Prop. *The Journal of Logic Programming* Vol. 23(3), pp.237–278, 1995.

[59] Van Hentenryck P., Cortesi, A. and Le Charlier, B., Type Analysis of Prolog Using Type Graphs, *The Journal of Logic Programming,* vol.22(3), pp. 179-208. (1995)

[60] Van Roy, P., and Despain, A. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer* 25(1): 54-68, 1992.

[61] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(4), 1992.