Combinations of Abstract Domains for Logic Programming

Agostino Cortesi[*]
Brown University
Box 1910, Providence
RI 02912 (USA)
aac@cs.brown.edu

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

Pascal Van Hentenryck[†]
Brown University
Box 1910, Providence
RI 02912 (USA)
pvh@cs.brown.edu

1 *Abstract*

Abstract interpretation [7] 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 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 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. The open product is general-purpose and can be used for other programming paradigms as well. *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(OProp$\otimes$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).

[*]Supported by CNR, Prog. Fin. "Sistemi Informatici e Calcolo Parallelo" under grant n.91.00026.69.

[†]Supported in part 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.

POPL 94- 1/94, Portland Oregon,USA

© 1994 ACM 0-89791-636-0/94/001..$3.50

2 *Introduction*

Abstract interpretation [7] 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 [27, 29]. Moreover, substantial work (e.g. [2, 10, 12, 14, 15, 18, 19]) has been devoted to produce efficient generic fixpoint algorithms and systems like GAIA[1] [19] 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 [3, 8, 9, 14, 25].

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 Pat($\Re$) for structural information.

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

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 form the environment). The open product improves on the *direct product* 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 [9]. It can be used as an alternative to the reduced product of the Cousots [8, 9] which cannot be automated, since it depends on the concretization functions, and which does not allow the components to interact. It contains as a degenerated case the refinement operation proposed independently by [3]. It also shares some of the motivations behind the ideas of $\mathcal{R}$-abstraction of [5] and open semantics [1], 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. Pat($\Re$) was motivated by the fact that structural information 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($\Re$). 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 [19, 23]. Note also that the motivations behind Pat($\Re$) are similar to those of [14] 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 which includes Pat(Prop), Pat(Type) and Pat(OProp⊗OMode⊗OPS), where Prop is the groundness domain of Marriott and Sondergaard [4, 22, 21], Type is the type graph domain of Bruynooghe and Janssens [16], and OMode and OPS are well-known domains for modes and sharing. It is interesting to note that Pat(OProp⊗OMode⊗OPS) and Pat(Type) are some of the most complicated domains ever implemented for Prolog, yet their requirements on the designer are minimal.

The rest of the paper is organized in four sections. Sections 2 and 3 present the main contributions of this paper, i.e. the open product and the generic pattern domain. Section 4 presents some experimental results and Section 5 concludes the paper. See the technical version of the paper for a comprehensive coverage [6].

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* is the simplest combination of abstract domains. Given two abstract domains D_1 and D_2 with concretization functions $Cc_1 : D_1 \rightarrow C$ and $Cc_2 : D_2 \rightarrow C$, 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 $\mathtt{OP}_c : C \rightarrow C$ 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 the direct product domain may contain many 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 more precise components.

The *reduced product* was proposed by the Cousots [8] 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 $\mathit{reduce} : D_1 \times D_2 \rightarrow D_1 \times D_2$ defined as

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

The reduced product domain is the domain

$$D = \{\mathit{reduce}(\langle d_1, d_2 \rangle) \mid d_1 \in D_1 \wedge d_2 \in D_2\}$$

while a reduced product operation over D can be defined as

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

Note that the reduced product 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]). However, the reduced product has also some inherent limitations. On the one hand, the implementation of the product operations cannot be obtained automatically, since they rely on the concretization function (a semantic notion), although the reduced product provides semantic guidance for their construction. On the other hand, as observed by the Cousots themselves, additional accuracy can be obtained by defining new operations where the operations on D_1 and D_2 interact. The reduced product has been used in logic programming by [3] and their reduced product operations are of course obtained manually.

The *open product* introduced in this section remedies the limitations of the reduced product. The key features of our approach are (1) an automatic derivation of the product operations; (2) possible interactions between the domains before, during, and after the product operation; (3) encapsulation of the representation and implementation of each component. As a consequence, the open product provides a rich and widely applicable framework to implement and prove the correctness of various combinations of domains, including the reduced product. It is also orthogonal to other systematic methods to build abstract domains such as downset completion and tensor products [8, 9, 25] 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. Note that all operations interact in terms of the initial abstract object, i.e. the object before executing the operation.

The open product can be improved further by letting the subdomains interact after the operations, i.e. in terms of the results of the operations. To capture formally this idea, the concept of refinement and refined open product are introduced. 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. Refinements are ways to compute or approximate the greatest lower bound operation used by the reduced product. In addition, they can be used even when the domains are not Galois connections or when the greatest lower bound is too expensive to compute exactly.

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. Finally, it is important to note that the open product is completely independent of logic programming and can be used for any programming language.

[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.

[3]We are using the term "abstract interpretation" in a technical sense here as in [5].

3.2 *Formalization*

In the following we assume familiarity with standard notions of abstract interpretation [7]. 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 an upper bound operation (not necessarily a lub operation).*

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 $Cc : D_2 \rightarrow D_1$ 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. The first important concept we introduce is the notion of query which gives information about the properties of concrete objects.

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

Definition 4 (query) *A query on the domain D is a monotone function $Q : D \rightarrow Arg \rightarrow Bool$ which maps elements of the domain D onto tests.*

Queries give rise to the notion of query interpretations which is a slight generalization of the traditional notion of interpretation [5] and was proposed independently in [24] for other purposes. In the following we denote by $\mathbf{D}$ the tuple

$$(D \ , \ \leq \ , \ \langle \mathtt{OP}_1, \ldots, \mathtt{OP}_n \rangle \ , \ \langle Q_1, \ldots, Q_m \rangle \ )$$

and by $\mathbf{D}_h$ the tuple

$$(D_h \ , \ \leq_h \ , \ \langle \mathtt{OP}^h_1, \ldots, \mathtt{OP}^h_n \rangle \ , \ \langle Q^h_1, \ldots, Q^h_m \rangle \ )$$

for $h = 0, 1, 2$.

Definition 5 (query interpretation) *A query interpretation 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 $\mathtt{OP}_i : D \rightarrow D$;[4] $Q_1 \ldots, Q_m$ are queries on D.*

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

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. 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) *An open operation on the domain D is a function $\mathtt{OP} : (Arg \rightarrow Bool)^m \rightarrow D \rightarrow D$ $(m \geq 0)$ which maps a tuple of tests onto an operation.* $\mathtt{OP}$ *should be monotone with respect to the tests.*

Definition 7 (open interpretation) *An open interpretation 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; and $\mathcal{Q}_1 \ldots, \mathcal{Q}_m$ are queries on D.*

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 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 $\forall d \in D_2 : \mathcal{Q}_1(Cc(d)) \Leftarrow \mathcal{Q}_2(d)$.*

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

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 $1 \leq i \leq n$: $\mathtt{OP}^2_i$ abstracts $\mathtt{OP}^1_i$, i.e.*

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

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 $\mathbf{D}_2$. 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 $\mathcal{Q}_i(d_1, d_2) = \mathcal{Q}^1_i(d_1) \vee \mathcal{Q}^2_i(d_2)$.*
- *the operation $\mathtt{OP}_i : (D_1 \times D_2) \rightarrow (D_1 \times D_2)$ is defined by:*

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

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, 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 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.

1. Domain: Domain $D_1 \times D_2$ abstracts D_0 through the concretization function $Cc : D_1 \times D_2 \rightarrow D_0$ defined by $Cc(d_1, d_2) = Cc(d_1) \sqcap Cc(d_2)$. This function is monotone by composition of monotone functions.

2. Queries: Query $\mathcal{Q}_i$ abstracts $\mathcal{Q}^0_i$ since

$$d_0 \leq_0 Cc(d_1), Cc(d_2)$$

implies

$$\mathcal{Q}^0_i(d_0) \Leftarrow \mathcal{Q}^1_i(d_1), \mathcal{Q}^2_i(d_2)$$

and thus

$$\mathcal{Q}^0_i(d_0) \Leftarrow \mathcal{Q}_i(d_1, d_2).$$

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

$d_0 \leq_0 Cc(d_1), Cc(d_2) \Rightarrow$

$\mathtt{OP}^0_i(d_0) \leq_0 Cc(\mathtt{OP}^h_i(\langle \mathcal{Q}^0_1(d_0), \ldots, \mathcal{Q}^0_m(d_0)\rangle)(d_h))$
(D_h is an open abstraction of D_0)

$\mathtt{OP}^0_i(d_0) \leq_0 Cc(\mathtt{OP}^h_i(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)(d_h))$
($\mathcal{Q}_j$ is an abstraction of $\mathcal{Q}^0_j$)

$\mathtt{OP}^0_i(d_0) \leq_0 \sqcap_{1 \leq h \leq 2}\ Cc(\mathtt{OP}^h_i(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)(d_h))$
(by properties of $\sqcap$)

$\mathtt{OP}^0_i(d_0) \leq_0 Cc(\mathtt{OP}_i(d_1, d_2))$
(by definition of $\mathtt{OP}_i$).

Hence, by $Cc(d_1, d_2) \leq_0 Cc(d_1), Cc(d_2)$,

$$\mathtt{OP}^0_i(Cc(d_1, d_2)) \leq_0 Cc(\mathtt{OP}_i(d_1, d_2)). \qquad \blacksquare$$

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

Refined Open Product The open product enables operations to benefit from information from the other components in the state before the product operation. 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 [3].

Definition 12 (refinement) *Let $\mathbf{D}_0$ and $\mathbf{D}_1$ be a query interpretation and an open interpretation such that $\mathbf{D}_1$ is an open abstraction of $\mathbf{D}_0$. An operation* REFINE*: $(Arg \rightarrow Bool)^m \rightarrow D_1 \rightarrow D_1$ 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.*

1. $d_0 \leq_0 Cc(d_1) \Rightarrow$
   $d_0 \leq_0 Cc(\mathtt{REFINE}(\langle \mathcal{Q}_1^0(d_0), \ldots, \mathcal{Q}_m^0(d_0)\rangle)(d_1))$;

2. $\mathtt{REFINE}(\langle \mathcal{T}_1, \ldots, \mathcal{T}_m\rangle)(d_1) \leq_1 d_1$
   for any test $\mathcal{T}_1, \ldots, \mathcal{T}_m$.

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 REFINE$_1$ and REFINE$_2$ are defined in $\mathbf{D}_1$ and $\mathbf{D}_2$ respectively. The corresponding operation 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* REFINE$_1$ *and* REFINE$_2$ *are refinement functions for $\mathbf{D}_1$ and $\mathbf{D}_2$ with respect to $\mathbf{D}_0$. The refinement function* REFINE *on the open product $\mathbf{D}_1 \otimes \mathbf{D}_2$ is defined by:*

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

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* REFINE$_1$ *and* REFINE$_2$ *are refinement functions for $\mathbf{D}_1$ and $\mathbf{D}_2$ with respect to $\mathbf{D}_0$. The operation*

$$\mathtt{OP}_i : (D_1 \times D_2) \rightarrow (D_1 \times D_2)$$

can be defined as:

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

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. Note also that the refinement can be applied arbitrarily often if useful.

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 it Subst the set of all substitutions and $Subst_F$ the set of substitutions whose domain is F, a concrete domain CS_F is simply $\wp(PS_F)$. This domain is a complete lattice with respect to the set inclusion $\subseteq$.[6]

Operations The operations on the concrete domains depend from one framework to another. As shown in [13], 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 ($\Theta \in CS_F$):

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

Queries For simplicity, we consider only two queries,

$$\mathcal{C}\mathtt{-GROUND} : CS_F \rightarrow F \rightarrow Bool$$

and

$$\mathcal{C}\mathtt{-NOSHARING} : CS_F \rightarrow F \times F \rightarrow Bool$$

which provide information on groundness and sharing and are specified as follows:

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

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

3.3.2 *The Open Abstract Interpretation* OProp

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

Domain Prop$_F$ [4, 22, 21] is the poset of Boolean functions that can be represented by propositional formulas constructed from F, the Boolean truth values, and the logical connectives $\vee, \wedge, \leftrightarrow$ and ordered by implication. A truth assignment over F is a function $I : F \rightarrow Bool$. The value of a Boolean function f wrt a truth assignment I is denoted $I(f)$. The basic intuition behind the domain Prop$_F$ is that a substitution θ is abstracted by a Boolean function f over F iff, for all instances θ' of θ, the truth assignment I defined by "$I(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 Prop$_F$ is a function Cc : Prop$_F \rightarrow CS_F$ defined as follows:

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

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

[6]In the following, the notion of substitution composition is slightly non-standard and makes sure that $dom(\theta\sigma) = dom(\theta)$.

Queries In Prop_F, the queries are abstracted by the functions

$$\begin{array}{l}\texttt{OProp-GROUND}: \text{Prop}_F \rightarrow F \rightarrow \mathit{Bool} \\ \texttt{OProp-NOSHARING}: \text{Prop}_F \rightarrow F \times F \rightarrow \mathit{Bool}\end{array}$$

whose definitions are as follows

$$\begin{array}{l}\texttt{OProp-GROUND}(f)(x_i) \Leftrightarrow (f \rightarrow x_i) \\ \texttt{OProp-NOSHARING}(f)(x_i, x_j) \Leftrightarrow (f \rightarrow x_i) \vee (f \rightarrow x_j)\end{array}$$

Operation The unification can be abstracted as $\texttt{OProp-UNIF}(\texttt{GROUND})(f, x_i, x_j) =$

$$\begin{cases} f \wedge x_i \wedge x_j & \text{if } \texttt{GROUND}(x_i) \vee \texttt{GROUND}(x_j); \\ f \wedge (x_i \leftrightarrow x_j) & \text{otherwise .}\end{cases}$$

Refinement The refinement in Prop_F is simply the function $\texttt{OProp-REFINE}(\texttt{GROUND})(f) = f \wedge x_{i_1} \wedge \ldots \wedge x_{i_p}$, where $\{x_{i_1}, \ldots, x_{i_p}\} = \{x_i \in F \mid \texttt{GROUND}(x_i)\}$.

3.3.3 *The Open Abstract Interpretation* OPS

Domain The abstract domain OPS (inspired by the sharing component described in [19]) specifies the possible pair-sharing of variables between terms. The elements of $\texttt{OPS}_F$ 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 $ps(x_i, x_j)$ 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 $Cc : \texttt{OPS}_F \rightarrow CS_F$ is

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

Queries OPS supports the sharing query:

$$\texttt{OPS-NOSHARING}(ps)(x_i, x_j) \Leftrightarrow (x_i, x_j) \notin ps$$

and the ground query

$$\texttt{OPS-GROUND}(ps)(x_i) \Leftrightarrow (x_i, x_i) \notin ps.$$

Operation The unification is abstracted as

$$\texttt{OPS-UNIF}(\texttt{GROUND})(ps, x_i, x_j) = \tilde{ps}'$$

where ps' is defined as

$$ps \setminus \{(x_k, x_l) \mid k \in \{i,j\} \vee l \in \{i,j\}\}$$

if $\texttt{GROUND}(x_i) \vee \texttt{GROUND}(x_j)$ and as

$$\begin{array}{l} ps\, \cup \\ \{(x_i, x_j)\} \cup \\ \{(x_i, x_k) \mid ps(x_j, x_k)\} \cup \{(x_j, x_k) \mid ps(x_i, x_k)\} \cup \\ \{(x_k, x_l) \mid \exists k', l' : ps(x_{k'}, x_k) \,\&\, ps(x_{l'}, x_l) \,\&\, k', l' \in \{i, j\}\}\end{array}$$

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

Refinement The refinement exploits groundness information. Let $W = \{x_i \in F \mid \texttt{GROUND}(x_i)\}$ in

$$\texttt{OPS-REFINE}(\texttt{GROUND})(ps) = \\ \{(x_i, x_j) \mid ps(x_i, x_j) \,\&\, x_i \notin W \,\&\, x_j \notin W\}.$$

3.3.4 *The Open Product* $\texttt{OProp} \otimes \texttt{OPS}$

The open product $\texttt{OProp} \otimes \texttt{OPS}$ is defined in Figure 1.

4 *The Generic Pattern Domain* $\texttt{Pat}(\Re)$

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[7]. However, abstract domains preserving structural information are often an order of magnitude more complicated to design.

In this section, we define $\texttt{Pat}(\Re)$, a generic abstract domain 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 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, $\texttt{Pat}(\Re)$ is tailored to logic programming. However, approaches similar in spirit can be used for other programming languages as well.

The key intuition behind $\texttt{Pat}(\Re)$ is to represent information on some subterms occurring in a substitution instead of information on terms bound to variables only. More precisely, $\texttt{Pat}(\Re)$ 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 [17], although $\texttt{Pat}(\Re)$ 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 [19, 23] which can be viewed as an instance of $\texttt{Pat}(\Re)$ for some specific domains.

$\texttt{Pat}(\Re)$ 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

$$< t_1, \ldots, t_p >$$

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

The open product $\mathcal{D} = \texttt{OProp} \otimes \texttt{OPS} = (D, \leq, \mathcal{D}\texttt{-REFINE}, \mathcal{D}\texttt{-UNIF}, \langle \mathcal{D}\texttt{-GROUND}, \mathcal{D}\texttt{-NOSHARING}\rangle)$ is defined as:

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

Figure 1: The Open Product $\texttt{OProp} \otimes \texttt{OPS}$

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 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 Prop, 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). 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 $x_1, \ldots, x_n$ is a triple (frm, sv, ℓ) where frm is a partial function, sv is a total function, and ℓ is an element of an $\Re$-domain to be specified.

Pattern Component The pattern component is defined as in [19]. 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 $Cc : FRM_p \rightarrow \wp(ST_p)$:

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

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

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

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 $< t_1, \ldots, t_p >$. 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 $Cc : \Re_p \rightarrow \wp(ST_p)$.

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.[8]

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

The third operation, *constrained mapping*, is novel and generalizes many operations such as projection, renaming, and extension among others. It is motivated by one of the fundamental difficulties encountered when designing the operations of Pat($\Re$): 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[9] and the ordering relation on Pat($\Re$). The constrained mapping provides a uniform solution to this problem and simplifies dramatically the implementation of many abstract operations.

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. *$tr_2^{\#} \circ tr_1^{\#} = (tr_1 \circ tr_2)^{\#}$;*
3. *$\ell \leq_{\Re_{p_1}} \ell'$ implies $tr^{\#}(\ell) \leq_{\Re_{p_2}} tr^{\#}(\ell')$;*
4. *$\{\langle t_{tr(1)}, \ldots, t_{tr(p_2)}\rangle : \exists\langle t_1, \ldots, t_{p_1}\rangle \in Cc(l)\} \subseteq Cc(tr^{\#}(l))$ (consistency).*

[8] The other cases come for free through Pat($\Re$).

[9] Intersection is used for example in reexecution frameworks (e.g. [20]).

The intuition is as follows: an element of $\Re_p$ is a constraint over the set of tuples of the form $< t_1, \ldots, t_p >$. A function $tr : I_p \rightarrow I_{p'}$ 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. This intuition is formally captured by function $tr^{\#}$ which indicates how to transform an abstract object in $\Re_p$ by removing superfluous terms and duplicating some others. The ordering on domains must obviously be respected since new equal terms are added in the same way to all elements of the domain. The constrained mapping can be implemented in a generic way in terms of simpler operations (see Figure 2) 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.

The Domain Pat($\Re$) Let D be a finite set of variables. The set of abstract substitutions Pat($\Re$) is the subset of $FRM \times SV \times \Re$ satisfying 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 $Cc : \text{Pat}(\Re) \rightarrow \wp(CS_D)$ defined by

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

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[10]. 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($\Re$) 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. *$\forall x \in D : sv_1(x) = tr(sv_2(x))$;*
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, since distinct elements (corresponding to permutations of indices) may have the same concretization. Formally, the domain should be defined as the quotient of Pat($\Re$) by this equivalence relation (as in the reduced domain construction). In practice, it suffices to work with a canonical form and hence we will continue working on the abstract domain Pat($\Re$).

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

We show generic implementation of the constraint mapping in terms of simpler operations on the domain $\Re$. These operations can be provided by the designer instead of the constrained mapping and operations are required to be monotonic and consistent abstractions of the following concrete operations:

Projection: This operation projects out of term t_j and can be easily extended to sets of indices.

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

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

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

Duplication: This operation duplicates an element.

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

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

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

The generic implementation is defined as follows.

Implementation 1 *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 $\left\{ \begin{array}{ll} 1) & tr_1 \text{ is a permutation;} \\ 2) & tr(tr_1(j)) = i_j \text{ for } j \in I_{p_3}; \end{array} \right.$
$tr_2 : I_{p_1} \rightarrow I_{p_3}$ such that $tr_2(i_j) = j$ for $j \in I_{p_3}$

in

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

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.

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

Theorem 2 *Implementation 1 of the constrained mapping is consistent.*

Proof:

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

■

Figure 2: A Generic Implementation of the Constrained mapping

4.3 *Implementation of the Abstract Operations*

We now turn to the implementation of the $\Re$-domain operations. Space requirements preclude the presentation of all operations and hence only operation `UNION` will be presented. Operation `UNION` illustrates well the process of building `Pat`($\Re$) operations in terms of $\Re$-operations and the benefit of the constrained mapping to overcome the difficulty encountered for certain operations in presence of global domains. The unification operation is described at length in the technical report and follows more closely the traditional implementation of the domain `Pattern`[19, 23].

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

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

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

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

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

The number of indices in the abstract substitution produced by the `UNION` operation will be exactly the size of F, i.e. $p = \#F$, 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 $tr : F \rightarrow I_p$ to establish the relation between the old and the new indices of the corresponding subterms. We denote by $tr_1 : I_p \rightarrow I_{p_1}$ and $tr_2 : I_p \rightarrow I_{p_2}$ the functions mapping elements of I_p to I_{p_1} and I_{p_2} respectively. $tr_1(k) = i$ if there exists $(i,j) \in F$ such that $tr(i,j) = k$, and analogously $tr_2(k) = j$ if there exists $(i,j) \in F$ such that $tr(i,j) = k$

Implementation 2 *The operation* `UNION`(β_1, β_2) *produces an abstract substitution $\beta = (frm, sv, \ell)$ defined as follows.*

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

Operation `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 $\Re$-operations. The various components are then deduced independently. Note also the simplicity gained by the availability of the constrained mapping.

4.4 *Applications*

The simplest applications of `Pat`($\Re$) amount to upgrading a single domain. Examples are the domain `Pat(Prop)` for groundness analysis and the domain `Pat(Type)`, upgrading the rigid type graph of Bruynooghe and Janssens [16] for type analysis. `Pat(Prop)` produces perfectly accurate results for our suite of benchmarks[11], improving on the domain `Prop` 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 inferring automatically recursive and disjunctive types. For instance, the analysis of the block planning program from Sterling and Shapiro [26] with `Pat(Type)` produces the following (optimal) grammar describing the type of the planning result:

```
T1 ::= [] | cons(T2,T1)
T2 ::= to_block(ground,ground,ground) |
       to_table(ground,ground,ground)
```

The advantages of using `Pat`($\Re$) are twofold: on the one hand, `Pat`($\Re$) factorizes sure structural information, keeping the sizes of the type graphs smaller; on the one hand, *pag* takes care of all other information such as modes, same-value, and sharing. Hence, the design of `Type` is simplified.

Another applications of `Pat`($\Re$) consists in having $\Re$ as an open product, combining the two contributions of this work. The domains `Pat(OProp` $\otimes$ `OPS)` and `Pat(OProp` $\otimes$ `OPS` $\otimes$ `OMode)`, where `OMode` is a mode domain assigning to each subterms a mode from { **var**, **ground**, **ngv**, **novar**, **noground**, **gv**, **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 slighlty 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(OProp` $\otimes$ `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. Only a small fraction of the available results are given; see the technical version of this paper for complete tables. The results were obtained with `GAIA` [19], all domains being implemented in C and the system being run on a Sun SS30/10.

The Benchmarks The programs we use are hopefully representative of "pure" logic programs (i.e. without the use of dynamic predicates). 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

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

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 [26] and implements an alpha-beta search procedure. The program `press` is an equation-solver program taken from [26] 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 [28]. 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 [11] 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 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).

On the Development Effort We first give some ideas about the effort necessary to produce the sophisticated domain `OPAT(OProp⊗OMode⊗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 `Prop` component requires 1791 lines (1766 + 25). For this application, only 19% of the overall code needs to be supplied. Domain `OPAT(OMode⊗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` [19][12]) 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.

[12]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.

On the Importance of Open Operations 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 domain `OPAT(OMode⊗OPS)` is used 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. The accuracy results 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, 0.42% for groundness, 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 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. The efficiency results 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.

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 domain `OPAT(OMode⊗OPS)` for the experimental results in its standard version (denoted by `S`) and in a modified version (denoted `NR`) where no refinement operations are used. The accuracy results are the same for the inputs and differ only on sharing for the outputs. In this case, `NR` produces about 230% of the sharing of `S`. This indicates that refinement operations can improve substantially the sharing component on the output patterns and seem to be useful in general, although the gain seems much less dramatic than in the case of open operations. More importantly perhaps, the efficiency results indicate that `NR` is about 1.3 times slower in the average than `S`, indicating that refinement operations can also improve efficiency by reducing the number of iterations.

On the Overhead of the Approach We turn to the overhead of our approach in `OPAT(OMode⊗OPS)` compared to a direct implementation of our pattern domain [19]. Of course, both domains have exactly the same accuracy. 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⊗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 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, including the reduced product construction. 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.

7 *Acknowledgments*

Comments and suggestions from the reviewers helped improve the presentation.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Conceptual and Software Support for Abstract Domain Design: Generic Structural Domain and Open Product. Technical Report CS-93-13, CS Department, Brown University, 1993.

[7] 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.

[8] 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.

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

[10] S. Debray. Efficient Dataflow Analysis of Logic Programs. *JACM*, 39(4):949–984, 1992.

[11] 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.

[12] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and Their Experimental Evaluation. *Software Practice and Experience*, 23(4), April 1993.

[13] R. Giacobazzi, S. Debray, and G. Levi. A Generlized Semantics for Constraint Logic Programs. In *FGCS'92*, Tokyo, June 1992.

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

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

[16] G. Janssens and M. Bruynooghe. Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.

[17] T. Kanamori and T. Kawamura. Analysing Success Patterns of Logic Programs by Abstract Hybrid Interpretation. Technical report, ICOT, 1987.

[18] 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.

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

[20] 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.

[21] 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.

[22] K. Marriott and H. Sondergaard. Abstract Interpretation of Logic Programs: the Denotational Approach, June 1990. To appear in ACM Transaction on Programming Languages.

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

[24] A. Mycroft. 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.

[25] F. Nielson. Tensor Product Generalize the Relational Data Flow Analysis Method. In *Proceedings of the Fourth Hungarian Computer Science Conference*, 1985.

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

[27] A. Taylor. LIPS on MIPS: Results From a Prolog Compiler for a RISC. In *Seventh International Conference on Logic Programming (ICLP-90)*, Jerusalem, Israel, June 1990.

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

[29] P. Van Roy and A. Despain. High-Performance Computing with the Aquarius Compiler. *IEEE Computer*, 25(1), January 1992.