Conceptual and Software Support for Abstract Domain Design: Generic Structural Domain and Open Product

A. Cortesi[1]
B. Le Charlier[2]
P. Van Hentenryck[3]

Technical Report No. CS-93-13
April 1993

[1]Brown University Box 1910, Providence, RI 02912
[2]University of Namur 21 rue grandgagnage B-5000 Namur (Belgium)
[3]Brown University Box 1910, Provodence, RI 02912

Conceptual and Software Support for Abstract Domain Design: Generic Structural Domain and Open Product

A. Cortesi
Brown University
Box 1910, Providence
RI 02912 (USA)
aac@cs.brown.edu

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

P. Van Hentenryck
Brown University
Box 1910, Providence
RI 02912 (USA)
pvh@cs.brown.edu

Abstract

Abstract interpretation [11] is a systematic methodology to design static program analysis which has been studied extensively in the logic programming community. 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 and complicated. The purpose of this paper is to propose conceptual and software support for the design of abstract domains. It contains two main contributions: a generic pattern domain and the notion of open product of domains. *The generic pattern domain* `Pat(`$\Re$`)` automatically upgrades a domain $\Re$ (e.g. `Prop` [30]), with structural information yielding a more accurate domain (e.g. `Pat(Prop)`) while abstracting away structural information from the designer. 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 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 `OPAT(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).

1 Introduction

Abstract interpretation is a systematic methodology to develop static program analysis [11]. Abstract interpretation has raised much interest in the logic programming because of the need of optimizations in compilers to make them competitive with procedural languages. Much progress has been achieved in this area since the early work of Mellish [31], including frameworks [1, 4, 7, 22, 28, 29, 35] abstract domains [5, 8, 15, 19, 21, 32, 34, 36] and fixpoint algorithms [24, 25, 26].

Our 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. With the emergence of efficient abstract interpretation 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 and needs to address issues such as the handling of structural information, the combination of domains, and the tradeoff between accuracy and efficiency. Yet little research has addressed the important problem of supporting this task adequately. Notable exceptions are [12, 6, 18].

The purpose of this paper is to propose some conceptual and software support to build sophisticated abstract domains. It contains two main contributions:

- a generic pattern domain `Pat(`$\Re$`)`;

- an open product construct $\otimes$.

The *generic pattern domain* `Pat(`$\Re$`)` automatically upgrades a domain $\Re$ (e.g. `Prop`) to obtain a new domain combining $\Re$ and structural information (e.g. `Pat(Prop)`). The key idea behind `Pat(`$\Re$`)` is to provide an implementation of the abstract operations in terms of simple operations on the domain $\Re$. `Pat(`$\Re$`)` is a generalization of the pattern domain of [33, 24], whose main idea is to preserve information on subterms. The main benefit of `Pat(`$\Re$`)` is to abstract away structural information from the designers which can focus on the salient aspects of their analysis. The motivations behind `Pat(`$\Re$`)` is similar to those of [18] 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 *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 domains which provide 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 queries. The open product can be viewed as a practical and modular way of implementing or approximating the reduced product construction of the Cousots [12, 13]. It contains as a degenerated case the refinement operation proposed independently by [6]. It also shares some of the motivations behind the ideas of $\mathcal{R}$-abstraction of [10] and open semantics [3], although the technical detail and practical applications are fundamentally different.

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 if offers to abstract domain designers. *Simplicity* is achieved by abstracting

away structural information (e.g. the main functor and its arguments) and irrelevant details of the subdomains; Structural information is ignored in most proposals for abstract domains because of the inherent complexity it introduces. Moreover, integrating various components in a single domain is error-prone and difficult; as a consequence, our contributions are of primary importance to build complex abstract domains. *Modularity* comes from the fact that abstract domains can be viewed as abstract data types whose operations are the queries and the abstract operations and which receives queries to improve accuracy of their operations. Modularity simplifies 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 and tested on several abstract domains including `Pat(Prop)`, `Pat(OProp`$\otimes$`OPS)`, `OPat(OMode`$\otimes$`OPS)` and `OPat(OProp`$\otimes$`OMode`$\otimes$`OPS)`, where `OPat(`$\Re$`)` is the open version of the generic domain `Pat(`$\Re$`)`. Note that `OPat(OMode`$\otimes$`OPS)` is equivalent to the domain `Pattern` [24] and that the abstract domain `OPat(OProp`$\otimes$`OMode`$\otimes$`OPS)` is probably one of the most complicated domains ever implemented for Prolog, yet its requirement on the designer are minimal.

The rest of the paper is organized in four sections. Section 2 presents the preliminaries. Section 3 presents the generic pattern domain `Pat(`$\Re$`)` from its semantics to its implementation and shows an application to the domain `Prop`. Section 4 introduces and formalizes the concept of open product for integrating domains and applies it to a variety of applications. Section 5 reports the experimental results on the mentioned domains. Section 6 contains the conclusions of this research. Some parts of the paper are rather technical. In a first reading, the reader may want to skip section 3.3 and section 4.4.

2 Preliminaries

In this section, we briefly review some basic definitions necessary for the rest of the paper. Since our contributions are essentially independent from the concrete semantics used (and hence from the abstract semantics and the fixpoint algorithm), we simply assume that the concrete semantics can be built from a reasonable set of operations on substitutions which were used in [23, 24, 26, 39, 40] to define various concrete semantics. The first subsection simply introduces the notion of substitution considered in this paper while the second subsection defines the concrete operations. The third subsection recalls some basic definitions on abstract interpretation.

2.1 Substitutions

We assume the existence of sets F_i and P_i $(i \geq 0)$ denoting sets of functors and predicate symbols of arity i. Let PV be an (infinite) set of *program variables*. Variables in PV are ordered and denoted by the $x_1, x_2, \ldots, x_i, \ldots$.

We assume the existence of another infinite set RV of *renaming variables*. We distinguish two kinds of substitutions: *program substitutions*, denoted θ, whose domain and codomain are subsets of PV and RV respectively, and *standard substitutions*, denoted σ, whose domain and codomain are subsets of RV. In the following, SP denotes the set of program substitutions, SP_D denotes the set of program substitutions the domain of which is D and SS the set of standard substitutions.

Let θ be a program substitution and $D \subseteq dom(\theta)$. The *restriction* θ' of θ to D, denoted $\theta_{/_D}$, is the substitution such that $dom(\theta') = D$ and $x_i\theta = x_i\theta'$ for all $x_i \in D$.

The definitions of *substitution composition* and *most general unifier* are the usual ones but are only used for standard substitutions. Program substitutions and standard substitutions can only be combined by applying a standard substitution σ to a program substitution θ. The result, denoted $\theta\sigma$, is defined by $dom(\theta\sigma) = dom(\theta)$ and $x(\theta\sigma) = (x\theta)\sigma$ for all $x \in dom(\theta)$. For program substitutions, the notion of *free variable* is non-standard to avoid clashes between variables during renaming. A free variable is represented by a binding to a renaming variable that appears nowhere else. Let Θ be a subset of SP. Θ is *complete* if and only if, for all $\theta \in \Theta$, θ and θ' are variant implies that $\theta' \in \Theta$.

Let D be a finite subset of PV. $CS_D = \{\Theta \subseteq SP : \forall\theta \in \Theta\, dom(\theta) = D$ and Θ is complete$\}$. CS_D is a complete lattice wrt set inclusion $\subseteq$.

2.2 Operations

We consider a number of operations on the concrete set of substitutions CS_D.

Union of Sets of Substitutions Let $\Theta_1, \ldots, \Theta_n \in CS_D$.

$\texttt{UNION}(\Theta_1, \ldots, \Theta_n) \;=\; \Theta_1 \cup \ldots \cup \Theta_n$.

Unification of two Variables Let $D = \{x_1, x_2\}$ and $\Theta \in CS_D$.

$\texttt{AI\_VAR}(\Theta) \;=\; \{\theta\sigma \mid \theta \in \Theta, \sigma \in SS,$ and $\sigma \in mgu(x_1\theta, x_2\theta)\}$

Unification of a Variable and a Functor This operation unifies x_1 with $f(x_2, \ldots, x_n)$. Let $D = \{x_1, \ldots, x_n\}$, $\Theta \in CS_D$, and $f \in F_{n-1}$.

$\texttt{AI\_FUNC}(\Theta, f) \;=\; \{\theta\sigma \mid \theta \in \Theta, \sigma \in SS,$ and $\sigma \in mgu(x_1\theta, f(x_2, \ldots, x_n)\theta)\}$

Restriction and Extension of a Set of Substitutions for a Clause The RESTRC operation restricts a set of substitutions on all variables in a clause to the variables in the head. The EXTC operation extends a set of substitutions on variables in the head of a clause to all variables in the clause. Let c a clause, D' be the set of variables in the head and D be the set of variables of c.

$\texttt{RESTRC}(c, \Theta) \;=\; \{\theta_{/_{D'}} \mid \theta \in \Theta\}$.
$\texttt{EXTC}(c, \Theta) \;=\; \{\theta \mid dom(\theta) = D,\ \theta_{/_{D'}} \in \Theta,$ and $\forall x \in D \setminus D',\ x$ is free in $\theta\}$.

Restriction and Extension of a Set of Substitutions for a Literal The RESTRG operation expresses a set of substitutions Θ in terms of the formal parameters $x_1, \ldots, x_n$ corresponding to a literal l. The EXTG operation extends a set of substitutions Θ with a set of substitutions Θ' representing the result of executing a literal l on Θ. Let D be the domain of Θ, $D'' = \{x_{i_1}, \ldots, x_{i_n}\}$ the set of variables appearing in l exactly in that order, and $D' = \{x_1, \ldots, x_n\}$.

$\texttt{RESTRG}(l, \Theta) \;=\; \{\theta \mid dom(\theta) = D', \exists\theta' \in \Theta : x_j\theta = x_{i_j}\theta'\ (1 \le j \le n)\}$.

$$\texttt{EXTG}(l,\Theta,\Theta') = \{\theta\sigma \mid \theta \in \Theta,\ \sigma \in SS\ ,\ \theta'\sigma \in \Theta'\ ,\ dom(\sigma) \subseteq codom(\theta'),\ (codom(\theta) \setminus codom(\theta')) \cap codom(\sigma) = \emptyset,\ dom(\theta') = D'\ ,\ x_j\theta' = x_{i_j}\theta\ (1 \leq j \leq n)\}.$$

2.3 Abstractions

In the following we assume familiarity with standard definitions and notations for abstract interpretation [11]. We assume for simplicity that all complete partial orders (cpo) are pointed.

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".*[1]

Definition 3 (abstraction of operations) [2] *Let D_1, D_2 be two domains, ordered by $\leq_1$ and $\leq_2$, respectively. Assume that D_2 abstracts D_1. Let $\texttt{OP}_1 : D_1 \rightarrow D_1$ and $\texttt{OP}_2 : D_2 \rightarrow D_2$. We say that $\texttt{OP}_2$ is an abstraction (or a consistent approximation) of $\texttt{OP}_1$ if $\forall d \in D_2 : \texttt{OP}_1(Cc(d)) \leq_1 Cc(\texttt{OP}_2(d))$.*

In the rest of the paper, we focus on domains which abstract the concrete domain of substitutions CS_D, and on operations which abstract the concrete operations `UNION`, ..., `EXTG` defined above.

3 The Generic Pattern Domain Pat(ℜ)

3.1 Overview

It is well-known that preserving structural information in abstract domains is often of primary importance to achieve a reasonable accuracy[3]. 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. The domain `Pat(`$\Re$`)` can be seen as a generalization of the domain `Pattern` defined in [33, 24]. Its main idea is to represent information about subterms occurring in a substitution. More precisely, `Pat(`$\Re$`)` may associate the following information with each considered subterm:

- its *pattern* which specifies the main functor of the subterm (if any) and the subterms which are its arguments;
- its properties which are left unspecified and are given in the domain $\Re$.

[1] Observe that we do not require the existence of a Galois connection between the domains [11]. This is due to the fact that, in practice, this requirement can be relaxed, without loss of correctness.

[2] We consider for simplicity only a unary operation. The generalization is straightforward.

[3] An alternative is to use reexecution which, in practice, simulates the presence of structural information although at a certain loss of accuracy.

In addition to the above information, each variable in the domain of the substitutions is associated with one of the subterms. Note that this information enables us to express that two arguments have the same value (and hence that two variables are bound together) by associating two arguments with the same subterm. To identify the subterms in an unambiguous way, an index is associated with each of them. If there are n subterms, we make use of indices $1, \ldots, n$. 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.

The main interest of `Pat(`$\Re$`)` is that it is generic with respect to the properties maintained on the subterms. `Pat(`$\Re$`)` provides the skeleton which contains structural and same-value information but leaves unspecified which information is maintained on the subterms. It is the role of the $\Re$-domain to specify this information, i.e. the $\Re$-domain describes properties of a set of tuples $< t_1, \ldots, t_p >$ where $t_1, \ldots, t_p$ are terms. Obviously, the $\Re$-domain does not need to be concerned with structural information, since this is the purpose of the generic domain. As a consequence, defining the $\Re$-domain mainly amounts to design a domain on clause variables, which is in general much simpler. The link between the subterms and the information stored in the domain is achieved by using indices. This will appear clearly in the next subsection.

`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. natural transformation) and a new implementation of some operations (e.g. `UNION`).

Notation In the following, we denote by I_p the set of indices $\{1, \ldots, p\}$, by ST_p the set of tuples of terms $< t_1, \ldots, t_p >$ and by ST the union of all sets ST_p for some $p \geq 0$. Finally, we denote by $\wp(ST)$ the powerset of ST.

3.2 Semantics

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. The

first two components form the "kernel" of the abstract domain, since they contain respectively the information about the terms corresponding to the program variables and about the structure of the terms.

3.2.1 Pattern Component

The pattern component 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 $\{i_1, \ldots, i_q\} \subset I_p$. The pattern component is a partial function $frm : I_p \not\to PAT_p$, where PAT_p is the set of all patterns on I_p, satisfying the following condition. Let G_{frm} be the graph whose nodes belong to I_p and whose arcs are the pairs (i, j) such that $frm(i) = f(\ldots, j, \ldots)$. The graph G_{frm} must be acyclic. We take the convention of denoting by $frm(i) = undef$ the fact that no pattern is associated with i. In the following, we denote by FRM_p the set of all 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 that specifies that the component represents all p_tuples of terms that satisfy simultaneously all pattern constraints:

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

The condition on G_{frm} ensures that $Cc(frm)$ is not empty.

3.2.2 Same Value Component

The second component assigns a subterm to each variable in the substitution. Given a set D of program variables and a set of indices I_m, this component is a surjective function $sv : D \to I_m$. 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 that makes sure that two variables assigned to the same index have the same value:

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

3.2.3 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 tuples are called $\Re$-tuples in the following. The domain is assumed to satisfy the requirements of definition 1. The signature of the concretization function Cc is as follows:

$$Cc : \Re_p \to \wp(ST_p)$$

In the following, we denote by $\Re$ the union of all $\Re_p$ $(p \geq 0)$.

The generic domain provides a standard implementation of the abstract operations `UNION`, `AI_VAR`, `AI_FUNC`, `RESTRC`, `EXTC`, `RESTRC` and `EXTG` in terms of a number of basic operations on the $\Re$-domain: $\Re$-`REN`, $\Re$-`INTR`, $\Re$-`PROJ`, $\Re$-`UNION`, $\Re$-`SPECAT`, $\Re$-`COMB`, $\Re$-`UNIF`, and $\Re$-`JOIN`. The rest of this section specifies the concrete versions of these operations, i.e. $\mathcal{C}$-`REN`, $\mathcal{C}$-`INTR`, $\mathcal{C}$-`PROJ`, $\mathcal{C}$-`UNION`, $\mathcal{C}$-`SPECAT`, $\mathcal{C}$-`COMB`, $\mathcal{C}$-`UNIF`, and $\mathcal{C}$-`JOIN`. According to definition 2, the abstract versions must be consistent approximations of the concrete operations.

Before specifying the operations, let us explain informally why some of them are needed. Operation $\Re$-INTR is used each time new terms are being introduced in a substitution. This is the case at clause entry (operation EXTC) as well as during the unification operations (operations AI_VAR, AI_FUNC, EXTG). Operation $\Re$-PROJ is used each time some terms should be removed from a substitution. This occurs in many operations, including clause exit (RESTRC) and procedure entry (RESTRG). Operations $\Re$-SPECAT, $\Re$-COMB, and $\Re$-UNIF are needed for various phases of the unification process. The implementation of the abstract unification mimics a recursive concrete unification as long as at least one of the patterns is known. The base case occurs when none of the patterns are known and this is when operation $\Re$-UNIF is called. The case of the two terms with compatible patterns is simply dealt with through recursion. The case when only one pattern, say the pattern of t_j, is known is reduced to the above recursive case by applying first to the other term, say t_i, a specialization operation which unifies t_i with a functor $f(y_1, \ldots, y_n)$ where $y_1, \ldots, y_n$ are new variables. Operation $\Re$-SPECAT is used to propagate the effect of this unification on the $\Re$-domain. Operation $\Re$-COMB is used by AI_FUNC before applying the general abstract unification algorithm. AI_FUNC simply constructs a new term using functor f and indices $i_1, \ldots, i_n$, and operation $\Re$-COMB is responsible to update the $\Re$-domain with the unification of this term and a newly introduced variable. Operation $\Re$-JOIN is used to concatenate two $\Re$-tuples in operation EXTG just before calling the general unification algorithm.

We are now in position to specify the concrete version of $\Re$-domain operations. Let Φ denote a subset of ST.

Introduction of variables: This operation introduces k variables in locations $m+1, \ldots, m+k$.

$$\mathcal{C}\text{-INTR}(\Phi, m, k) = \{ < t_1, \ldots, t_m, y_1, \ldots, y_k, t_{m+1}, \ldots, t_p > \mid < t_1, \ldots, t_p > \in \Phi \ \& \ y_1, \ldots, y_k \text{ are new distinct variables } \}.$$

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.

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

Union: This operation takes the union of two sets of tuples.

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

Unification Operations: We now turn to the unification operations.

$$\begin{aligned} \mathcal{C}\text{-SPECAT}(\Phi, i, \{j_1, \ldots, j_n\}, f) \ = \ \{ < t_1\sigma, \ldots, t_p\sigma > \mid \ & < t_1, \ldots, t_p > \in \Phi \ \& \\ & \sigma \in mgu(t_i, f(t_{j_1}, \ldots, t_{j_n})) \ \& \\ & \sigma \in SS \ \& \\ & t_{j_1}, \ldots, t_{j_n} \text{ are distinct free variables } \}. \end{aligned}$$

$$\mathcal{C}\text{-COMB}(\Phi, i, \{j_1, \ldots, j_n\}, f) = \{ < t_1\sigma, \ldots, t_p\sigma > \mid \begin{array}{l} < t_1, \ldots, t_p > \in \Phi \ \& \\ \sigma \in mgu(t_i, f(t_{j_1}, \ldots, t_{j_n})) \ \& \\ \sigma \in SS \ \& \\ t_i \text{ is a free variable } \}. \end{array}$$

$$\mathcal{C}\text{-UNIF}(\Phi, i, j) = \{ < t_1\sigma, \ldots, t_p\sigma > \mid < t_1, \ldots, t_p > \in \Phi \ \& \ \sigma \in mgu(t_i, t_j) \ \& \ \sigma \in SS \ \}.$$

Join: This operation concatenates tuples of terms coming from two different sets

$$\mathcal{C}\text{-JOIN}(\Phi_1, \Phi_2) = \{< t_1^1, \ldots, t_n^1, t_1^2, \ldots, t_m^2 > \mid < t_1^1, \ldots, t_n^1 > \in \Phi_1 \ \& \ < t_1^2, \ldots, t_m^2 > \in \Phi_2 \ \}.$$

Renaming of Variables: The $\Re$-domain also needs a renaming operation. Let $r : I_p \to I_p$ be a renaming of indices.

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

3.2.4 The Concretization Function

We are now in position to specify the domain Pat($\Re$).

Definition 4 *Let D be a finite set of program 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) $\forall i : m < i \leq p : \exists j : 1 \leq j \leq p : frm(j) = f(\ldots, i, \ldots)$.

Formally, the meaning of an abstract substitution $\beta = (frm, sv, \ell)$ is given by the following concretization function:

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

It remains to define the ordering relation on substitutions. 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[4]. Conceptually, $\beta_1 \leq \beta_2$ should hold iff β_1 imposes the same or more constraints on all components than β_2 does, that is iff $Cc(\beta_1) \subseteq Cc(\beta_2)$. To obtain a syntactic definition, we assume the existence of the following notion on the domain $\Re$.

Definition 5 (Natural Transformation) *A* natural transformation *on domain $\Re$ is a law (i.e. a* functor *in the terminology of Category theory) which maps any function $tr : I_{p_2} \to I_{p_1}$ onto a function $tr^{\#} : \Re_{p_1} \to \Re_{p_2}$. This law has to satisfy the following conditions:*

1. $id^{\#}_{I_p}(\ell) = \ell$;[5].

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

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

The intuition is as follows: an element of $\Re_p$ 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 above explanation also justifies that $tr^{\#}$ will always exist for reasonable domains and will be natural. We are now ready to present the ordering on the abstract substitutions.

Definition 6 $\beta_1 \leq \beta_2$ *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. $\forall i \in I_{p_2} : frm_2(i) = f(i_1, \ldots, i_q) \Rightarrow frm_1(tr(i)) = f(tr(i_1), \ldots, tr(i_q))$.

It is important to note here 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[6]. In practice, there is no advantage in proceeding in this way and we will continue working on the abstract domain `Pat(`$\Re$`)`.

Note that the natural transformation can be implemented in two ways:

1. a specific way: the $\Re$-domain is required to define the transformation; this will be in general the most efficient solution, since locality can be exploited if necessary;

2. a generic way: the transformation can be implemented in a standard way by using more basic operations required on the domain.

In the next section, we describe the generic implementation.

3.3 Implementation: Abstract Operations

It remains now to specify the generic abstract operations on the above abstract domain, namely `UNION, AI_VAR, AI_FUNC, RESTRC, EXTC, RESTRC` and `EXTG`. For each of these operations we provide the specification and a safe implementation. In many cases, the correctness proofs are direct consequences of the specifications and can be fully described as a formal generalization of the corresponding proofs in [33] (also sketched in [24]). This section is quite technical and can be skipped at first reading.

[6] This is similar to the notion of reduced domain in [13].

Notation: In what follows, we denote by β an abstract substitution $(frm, sv, \ell) \in \mathtt{Pat}(\Re)$ with $frm \in FRM_p$, $sv \in SV_{D,m}$ and $\ell \in \Re_p$. Observe, in particular, that β marks out the indices p and m.

3.3.1 The Natural Transformation

As mentioned previously, the natural transformation can be implemented in a standard way using three more basic operations which are required to be monotone: $\Re$-REN, $\Re$-PROJ, and $\Re$-EQU. The last operation can be obtained from $\Re$-INTR and $\Re$-UNIF but we provide its specification for clarity. Its main purpose is to introduce an equality constraint between an old and a new index.

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

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

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

We are now in position to define the natural transformation in a generic way.

Implementation 1 *The natural transformation $tr^{\#}$ of $tr : I_{p_2} \rightarrow I_{p_1}$ can be defined as follows. Let*

- $p_3 = \#tr(I_{p_2})$;[7]
- $i_1, \ldots, i_{p_3}$ *be such that* $i_1 < \ldots < i_{p_3}$ *and* $\{i_1, \ldots, i_{p_3}\} = tr(I_{p_2})$;
- $tr_1 : I_{p_2} \rightarrow I_{p_2}$ *such that*
  1. tr_1 *is a permutation;*
  2. $tr(tr_1(j)) = i_j$ *for* $j \in I_{p_3}$;
- $tr_2 : I_{p_1} \rightarrow I_{p_3}$ *such that* $tr_2(i_j) = j$ *for* $j \in I_{p_3}$

in

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

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.

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

[7] We use $\#A$ to denote the cardinality of a set A.

Theorem 1 *Implementation 1 of the natural transformation is consistent.*

Proof:

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

■

3.3.2 Operation EXTC$(\beta, c) = \beta'$

This operation extends a substitution with new variables at the entry of a clause.

Specification 1 *Let $D = \{x_1, \ldots, x_n\}$ be the set of variables in the head of c, $dom(\beta) = D$ and $D' = \{x_1, \ldots, x_{n+k}\}$ be the set of all variables in c.* EXTC(β, c) *produces a substitution $\beta' = (frm', sv', \ell')$ such that*

$$
\begin{array}{l}
Cc(\beta') = \{\theta \mid \ dom(\theta) = D' \ \ \& \ \ \theta_{/D} \in Cc(\beta) \ \ \& \ \ x_{n+1}\theta, \ldots, x_{n+k}\theta \text{ are distinct renaming variables} \\
\qquad\qquad \ \& \ \ x_{n+j}\theta \notin codom(\theta_{/D}) \ (1 \leq j \leq k)\}.
\end{array}
$$

Implementation 2 *Under the assumptions of Specification 1,* EXTC$(\beta, c) = \beta'$, *where:*

$$
\begin{array}{lcl}
p' & = & p + k. \\
m' & = & m + k.
\end{array}
$$

$$
r(i) \ = \ \begin{cases} i & 1 \leq i \leq m; \\ i - k & m' < i \leq p'. \end{cases}
$$

$$
frm' \ = \ \{\langle i, f(i_1, \ldots, i_n)\rangle \mid \langle r(i), f(r(i_1), \ldots, r(i_n))\rangle \in frm\}
$$

$$
\begin{array}{lcl}
sv'(x_i) & = & sv(x_i) \ \ (1 \leq i \leq n). \\
sv'(x_{n+i}) & = & m + i \ \ (1 \leq i \leq k).
\end{array}
$$

$$
\ell' \ = \ \Re\text{-INTR}(\ell, m, k).
$$

The implementation is straightforward. To accommodate the formal structure of abstract substitutions, the new variables must be mapped onto indices $m + 1, \ldots, m + k$. Hence, the function r is introduced to shift the old indices greater than m.

3.3.3 Operation RESTRC$(\beta, c) = \beta'$

This operation restricts the output substitution of a clause to the head variables.

Specification 2 *Let D be the domain of β and D' be the set of variables in the head of c ($D' \subseteq D$).* `RESTRC`(β, c) *produces a substitution $\beta' = (frm', sv', \ell')$ such that*

$$Cc(\beta') = \{\theta_{/D'} \mid \theta \in Cc(\beta)\}$$

This operation needs to compute the set of indices reachable from a given index as the other indices are no longer relevant.

Definition 7 *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. $\{i\} \cup \bigcup_{j=1}^{n} reachable(i_j, frm)$ *if* $frm(i) = f(i_1, \ldots, i_n)$.

We also use $reachable(I, frm)$ to denote $\bigcup_{i \in I} reachable(i, frm)$.

Implementation 3 *Under the assumptions of Specification 2,* `RESTRC`$(\beta, c) = \beta'$, *where*

$$\begin{array}{lcl} sv(D') & = & \{sv(x) \mid x \in D'\} \\ I & = & reachable(sv(D'), frm); \\ m' & = & \#sv(D'); \\ p' & = & \#I; \\ W & = & \{i \in I_p \mid i \notin I\}. \end{array}$$

r is the unique strictly increasing function from I to $I_{p'}$;

$$\begin{array}{lcl} frm' & = & \{\ \langle r(i), f(r(i_1), \ldots, r(i_n))\rangle \mid \langle i, f(i_1, \ldots, i_n)\rangle \in frm \text{ and } i \in I\ \}; \\ sv'(x) & = & r(sv(x)) \quad \forall x \in D'; \\ \ell' & = & \Re\text{-}\mathtt{PROJ}(\ell, W). \end{array}$$

The operation `RESTRG` can be deduced easily from the above operation and an operation that produces a variable renaming.

3.3.4 Unification

Overview The unification operation will be handled through a number of suboperations which significantly reduce the combinatorial explosion of subcases. The base case is operation `UNIF1`, which unifies two subterms whose patterns are undefined. Operation `SPECAT` is used to unify two terms, one with a pattern and the other without. The operation specializes the second term so that both have the same pattern and the general case applies. The general case is taken care of by operation `UNIF`. Operation `FCTA` simply removes one subterm after unification. The operations `AI_VAR, AI_FUNC` and `EXTG` can be derived from operation `UNIF` in a direct way.

A Note on Notation Consider the unification of two terms and observe that after a unification, only one of the terms is necessary and the other one can be removed. Moreover, the same value component should be updated to reflect the removal of the subterm. These operations are rather frequent in the unification process and, instead of updating permanently the components, the equalities will be stored using a function $fi : I_q \rightarrow I_p$ such that $fi(i) = fi(j) \Rightarrow t_i = t_j$. This allows us to simplify the presentation and the implementation as well[8]. 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 α-tuples associations of two components (frm, ℓ) where frm, ℓ are defined on the same set of indices I_p. α-tuples will be denoted in the following by α and α' possibly subscripted. Formally α defines a set of p_tuples of terms $Cc(\alpha)$ such that

$$Cc(\alpha) \;=\; Cc(frm) \cap Cc(\ell).$$

Moreover, we call a δ-tuple the association (α, fi) of an α-tuple α and a function fi. The remaining operations will be defined on δ-tuples. Note also that we will implicitly assume that a δ-tuple δ_k is associated to a pair (α_k, fi_k) (and similarly a δ-tuple δ' to a pair (α', fi')).

As usual, we define the meaning of δ-tuples by means of a concretization function as follows:

$$\begin{array}{rcl} Cc(\delta) & = & \{\langle u_1, \ldots, u_q\rangle \mid \exists \langle t_1, \ldots, t_p\rangle \in Cc(\alpha) : u_i = t_{fi(i)} \;\; (1 \leq i \leq q)\} \\ & = & \{\langle t_{fi(1)}, \ldots, t_{fi(q)}\rangle \mid \langle t_1, \ldots, t_p\rangle \in Cc(\alpha)\} \end{array}$$

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 $t : I_{p_2} \rightarrow I_{p_1}$. We take the convention of representing those expressions as

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

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

Fusion of δ-tuples 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 α-tuple α. The basic idea behind this operation is to remove a subterm which is no longer necessary.

Specification 3 *Let $\bar{u} = \langle u_1, \ldots, u_q\rangle$ 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 $max = max(i, j) \;\; (fi)$

1. $ti : I_p \rightarrow I_{p-1}$

$$\begin{array}{rcll} ti(k) & = & k & \text{if } k < max; \\ & = & min(i, j) \;\; (fi) & \text{if } k = max; \\ & = & k - 1 & \text{if } k > max; \end{array}$$

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

2. $it : I_{p-1} \rightarrow I_p$

$$\begin{array}{rcll} it(k) & = & k & \text{if } k < max; \\ & = & k+1 & \text{if } k \geq max. \end{array}$$

When applied to the α-tuple, 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 4 *The function* $\texttt{FCTA}(i, j, \delta)$ *is defined as*

$$\begin{array}{rcl} frm' & = & \{\langle ti(k), f(ti(k_1), \ldots, ti(k_n))\rangle \mid \langle k, f(k_1, \ldots, k_n)\rangle \in frm\}; \\ \ell' & = & \Re\text{-}\texttt{PROJ}(\ell, max); \\ fi' & = & ti \circ fi. \end{array}$$

Note that the function fi of the δ-tuple needs to be updated as well. The operation FCTA can be easily extended to sets of pairs.

Operation $\texttt{UNIF1}(i, j, \delta) = \delta'$

Specification 4 *This operation is defined on an* δ*-tuple* $\delta = (\alpha, fi)$*, where* $\alpha = (\ell, frm)$*, assumes that* $frm(i) = frm(j) = undef\ (fi)$*, and produces another* δ*-tuple* $\delta' = (\alpha', fi')$*, where* $\alpha' = (\ell', frm')$*. Informally speaking,* $\texttt{UNIF1}(i, j, \delta)$ *unifies subterms* i *and* j *in the* δ*-tuple* δ*, giving* δ'*.*

More formally, given $\bar{t} = \langle t_1, \ldots, t_p\rangle$*, a p_tuple of terms, and* σ *a substitution, 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. It performs the unification of u_i and u_j (i.e. of $t_{fi(i)}$ and $t_{fi(j)}$), using operation UNIF1 on α-tuples. Then FCTA is used to remove one of the redundant terms.

Implementation 5 *Let* $\texttt{UNIF1}(i,j,\alpha) = \alpha'$ *be defined as follows: the pattern component remains the same as no new pattern is introduced, while the* $\Re$ *component is obtained by adding to* ℓ *the matching between* t_i *and* t_j*.*

$\texttt{UNIF1}(i, j, \alpha) = \alpha'$ *where*

$$\begin{array}{rcl} frm' & = & frm \\ \ell' & = & \Re\text{-}\texttt{UNIF}(\ell, i, j) \end{array}$$

$\texttt{UNIF1}(i, j, \delta) = \delta'$ *where*[9]

$$\begin{array}{rcl} \delta' & = & \texttt{FCTA}(i, j, \delta_1) \\ \alpha_1 & = & \texttt{UNIF1}(i, j, \alpha)\ (fi) \\ fi_1 & = & fi \end{array}$$

[9]Recall here that $\delta_1 = (\alpha_1, fi_1)$.

3.3.5 Operation 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 SPECAT performs the first step. The second step is carried out by the general unification procedure.

Specification 5 *We first specify* SPECAT *on α-tuple and then we extend the definition to δ-tuple. Given $\bar{t} = \langle t_1, \ldots, t_p \rangle$, a p_tuple of terms, σ a substitution, $y_1, \ldots, y_n$, n distinct variables not occurring in $\bar{t}$, the operation* SPECAT$(i,j,\alpha) = \alpha'$ *verifies*

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

Implementation 6 *The implementation returns $\alpha' = \perp$ 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 $\{t_{p+1}, \ldots, t_{p+n}\}$, whereas the pattern component is defined by adding the new pattern :*

$$\begin{array}{lcl} frm' & = & frm \cup \{\langle i, f(p+1, \ldots, p+n)\}; \\ \ell' & = & \Re\text{-SPECAT}(\Re\text{-INTR}(\ell, p, n), i, \{(p+1), \ldots, (p+n)\}, f). \end{array}$$

Operation SPECAT can be adapted to δ-tuples. The specification is the same as before except that α-tuples are replaced by δ-tuples. The implementation simply uses SPECAT for α-tuples and build a new function fi' to take into account the new indices in α'.

Implementation 7 *Assume that $\delta = (\alpha, fi)$ with $dom(fi) = q$ and $1 \leq i, j \leq q$. Operation* SPECAT$(i,j,\delta) = \delta'$ *where*

$\delta' = \perp$ *if* SPECAT(i,j,α) $(fi) = \perp$

$\delta' = (\alpha', fi')$ *otherwise*
with

$\alpha' =$ SPECAT$(i,j,\alpha)(fi)$
$fi' : I_{q+n} \rightarrow I_{p+n}$ *is such that*
$fi'(k) = fi(k)$ *if* $k \leq q$
$fi'(k) = k - q + p$ *if* $k > q$.

3.3.6 Operation UNIF$(i,j,\delta) = \delta'$

Let us present the main procedure for unification UNIF$(i,j,\delta) = \delta'$. 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 $\langle u_1, \ldots, u_m \rangle$ is a prefix of $\langle t_1, \ldots, t_n \rangle$ $(m \leq n)$ iff $u_i = t_i$ $(1 \leq i \leq m)$.

Specification 6 *Given $\bar{u} = \langle u_1, \ldots, u_q \rangle$, 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 8 *The implementation of operation* UNIF *is as follows:*

if $i = j \quad (fi)$ **then**
$\delta' = \delta$

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

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

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

$$\delta_\circ = \left\{\begin{array}{ll} \texttt{SPECAT}(i, j, \delta) & \textbf{if } (frm(i) = undef \quad (fi)) \\ \texttt{SPECAT}(j, i, \delta) & \textbf{if } (frm(j) = undef \quad (fi)) \\ \delta & \textbf{otherwise} \end{array}\right.$$

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

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

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 [33].

3.3.7 Operation AI_VAR(β) = β'

This operation unifies two variables x_1 and x_2 in an abstract substitution β. This operation simply unifies subterms 1 and 2 to obtain a new δ-tuple. The α-tuple produces the pattern component and the $\Re$-component, while the same-value component is obtained from the old same-value component and the function of the δ-tuple.

Specification 7 *Let β such that $dom(\beta) = \{x_1, x_2\}$. The following holds for any program substitution θ and any renaming substitution σ:*

$$\theta \in Cc(\beta) \,\&\, \sigma \in mgu(x_1\theta, x_2\theta) \quad \Rightarrow \; \theta\sigma \in Cc(\beta').$$

Implementation 9 *The implementation of* AI_VAR(β) *with $\beta = (sv, \alpha)$ produces a substitution $\beta' = (sv', \alpha')$ with*

$$(\alpha', fi') = \texttt{UNIF}(sv(x_1), sv(x_2), (\alpha, id)) \text{ and } sv' = fi' \circ sv$$

where id denotes the identity function.

3.3.8 Operation AI_FUNC$(f, \beta) = \beta'$

This abstract operation unifies a variable x_1 with a term $f(x_2, \ldots, x_n)$ in an abstract substitution β. It is achieved by creating a new subterm $p+1$ containing the new function to unify and unifying terms $p+1$ and 1.

Specification 8 *Let β such that $dom(\beta) = \{x_1, \ldots, x_n\}$. The following holds for any program substitution θ and any renaming substitution σ:*

$$\theta \in Cc(\beta) \;\&\; \sigma \in mgu(x_1\theta, f(x_2, \ldots, x_n)\theta) \;\; \Rightarrow \; \theta\sigma \in Cc(\beta').$$

Implementation 10 *Assuming that $i_k = sv(x_k)$ $(1 \leq k \leq n)$, the implementation of* AI_FUNC(β) *produces (sv', α') where*

$$\begin{array}{lcl} (\alpha', fi') & = & \texttt{UNIF}(i_1, p+1, \delta_1) \\ frm_1 & = & frm \cup \{\langle p+1, f(i_2, \ldots, i_n)\rangle\} \\ \ell_1 & = & \Re\texttt{-COMB}(\Re\texttt{-INTR}(\ell, p, 1), p+1, \{i_2, \ldots, i_n\}, f) \\ fi_1 & = & id \\ sv' & = & fi' \circ sv. \end{array}$$

3.3.9 Operation EXTG$(g, \beta_1, \beta_2) = \beta'$

This operation is used to extend a clause substitution (i.e. β_1) with the result of the execution of a procedure call (i.e. β_2).

Specification 9 *Let $x_{i_1}, \ldots, x_{i_k}$ be the sequence of variables occurring in the procedure call g. Let us define $D = \{x_{i_1}, \ldots, x_{i_k}\}$. Let β_1 be an abstract substitution such that $\{x_{i_1}, \ldots, x_{i_k}\} \subseteq \{x_1, \ldots, x_m\} = dom(\beta_1)$. Let β_2 be an abstract substitution such that $\{x_1, \ldots, x_k\} = dom(\beta_2)$. Let θ_1 and θ_2 such that $dom(\theta_i) = dom(\beta_i)$ $(i = 1, 2)$. Let σ a renaming substitution. The following holds:*

$$\left.\begin{array}{l} \theta_1 \in Cc(\beta_1) \;\&\; \theta_2 \in Cc(\beta_2) \\ (\forall j : 1 \leq j \leq k : x_j\theta_2 = x_{i_j}\theta_1\sigma) \\ dom(\sigma) \subseteq codom(\theta_{1/D}) \\ (codom(\theta_1) - codom(\theta_{1/D})) \cap codom(\sigma) = \emptyset \end{array}\right\} \Rightarrow \; \theta_1\sigma \in Cc(\beta').$$

We make use of a procedure UNIF_LIST$(List, \delta)$ whose input is a list of pairs $(i_1, j_1), \ldots, (i_n, j_n)$ and a δ-tuple, and whose output is another δ-tuple where the terms i_k, j_k have been unified. (It is simple to define this procedure from procedure UNIF or to generalize UNIF to obtain it).

The implementation proceeds in two steps. First it builds a unique term that contains both substitutions. Then, it unifies the corresponding arguments of the substitutions.

Implementation 11 *Let $x_{i_1}, \ldots, x_{i_k}$ be the sequence of variables occurring in g and assume that*

$\delta = (\alpha, id)$ *where*

$\alpha = (\ell, frm)$

$\ell = \Re\texttt{-JOIN}(\ell_1, \ell_2)$

$frm = frm_1 \cup \{\langle i + p_1, f(i_1 + p_1, \ldots, i_n + p_1)\rangle \mid \langle i, f(i_1, \ldots, i_n)\rangle \in frm_2\}$

EXTG(g, β_1, β_2) *produces* $\beta' = (sv', \alpha')$ *where*

$(\alpha', fi') =$ UNIF_LIST(l, δ)
$sv' = fi' \circ sv_1$
$l = ((sv_1(x_{i_1}), sv_2(x_{i_1}) + p_1), \ldots, (sv_1(x_{i_k}), sv_2(x_{i_k}) + p_1))$.

3.3.10 Operation UNION$(\beta_1, \beta_2) = \beta$

Specification 10 *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*

$$dom(\beta) = D \quad \& \quad Cc(\beta_1) \cup Cc(\beta_2) \subseteq Cc(\beta).$$

or, when $\Re$-UNION *computes an upper bound,*

$$dom(\beta) = D \quad \& \quad \beta_1, \beta_2 \leq \beta$$

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 components. Hence 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 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 substitutions. Of course, the number of variables n is the same in β, β_1, β_2. We also need a bijective function $tr : F \to I_p$ 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 $(i, j) \in F$ 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 12 *The operation* UNION(β_1, β_2) *produces a substitution* $\beta = (frm, sv, \ell)$ *defined as follows.*

UNION$(\beta_1, \beta_2) = \beta$

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

3.4 Application: The Domain Pat(Prop)

In this section, we present the domain Pat(Prop), as an instantiation of the generic domain Pat($\Re$) to the domain Prop [30, 9, 27]. Pat(Prop) is particularly appropriate for groundness analysis, although there are many other possible uses of boolean formulas[7, 2]. Prop represents groundness information in terms of propositional variables and a number of logical connectives ($\wedge, \vee, \leftrightarrow$). For simplicity, we use the integers $1, 2, \ldots, i, \ldots$ as propositional variables, since they refer directly to the subterms they are associated with. Readers which are not comfortable with this notation can simply see them as an abbreviation for a set of variables $w_1, w_2, \ldots, w_i, \ldots$.

3.4.1 Domain

An element of $\texttt{Prop}_p$ is the poset of boolean functions that can be represented by propositional formulas constructed from I_p, the logical connectives $\wedge, \vee$ and $\leftrightarrow$, and the boolean constants true and false and is ordered by implication. It is easy to see that $\texttt{Prop}_p$ is a finite lattice where the greatest lower bound is given by conjunction and the least upper bound by disjunction.

Definition 8 *A truth assignment over I_p is a function $T : I_p \rightarrow Bool$. The value of a boolean function f wrt a truth assignment T is denoted $T(f)$. When $T(f) =$ true, we say that T satisfies f.*

The basic intuition behind the domain Prop is that a tuple of terms $< t_1, \ldots t_p >$ is abstracted by a Boolean function f iff, for all instances $< t'_1, \ldots t'_p >$ of $< t_1, \ldots t_p >$, the truth assignment T defined by

$$T(i) = \text{true iff } t'_i \text{ is ground } (1 \leq i \leq p)$$

satisfies f.

For instance, $1 \leftrightarrow 2$ abstracts the tuples of terms $< y_1, y_1 >$, $< a, a >$, but not $< a, y_2 >$ nor $< y_1, y_2 >$.

Definition 9 *The concretization function for $\texttt{Prop}_p$ is a function $Cc : \texttt{Prop}_p \rightarrow \wp(ST_p)$ defined as*

$$Cc(f) = \{< t_1, \ldots, t_n > \mid \forall \sigma \in SS : \texttt{assign}\ (< t_1\sigma, \ldots, t_p\sigma >)(f) = \texttt{true}\}$$

where $\texttt{assign} : ST_p \rightarrow I_p \rightarrow Bool$ *is defined by* $\texttt{assign} < t_1, \ldots, t_p > (i) =$ true *iff t_i is ground.*

The following definitions will be used later.

Definition 10 *The valuation of a function f wrt a variable i and a truth value b, denoted $f|_{i=b}$, is the function obtained by replacing i by b in f.*

Definition 11 *The denormalization of a formula f wrt $[i_1, \ldots, i_n]$ is the boolean function obtained by replacing simultaneously $1, \ldots, n$ by $i_1, \ldots, i_n$ in f. This denormalization is denoted* $\texttt{denorm}\ [i_1, \ldots, i_n]\ f$.

3.4.2 Operations

The implementations of the operations required by the generic domain Pat(ℜ)are conceptually simple in the case of Prop.

Renaming of a formula: Let $r : I_p \to I_p$ be a renaming of indices, i.e. a bijective function from the set $\{1, \ldots, p\}$ to itself. The Prop-REN operation is implemented by using the denorm function previously defined.

$\texttt{Prop-REN}(f, r) = \texttt{denorm}\ [r(1), \ldots, r(p)]\ f.$

Introduction of new variables: $\texttt{Prop-INTR}(f, m, k)$ is obtained by shifting the indices of the last $p - m$ variables by k positions.

$\texttt{Prop-INTR}(f, m, k) = \texttt{denorm}\ [1, \ldots, m, m + 1 + k, m + 2 + k, \ldots, p + k]\ f.$

Projection of a variable: Projection is simply defined as:

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

Union of abstract tuples: The least upper bound in Prop of two propositional formulas f_1, f_2 over the same variables corresponds to their logical disjunction.

$\texttt{Prop-UNION}(f_1, f_2) = f_1 \vee f_2.$

Note that the least upper bound does not lose precision in Prop.

Unification of a variable and a functor: The two specializations ℜ-SPECAT and ℜ-COMB of the unification of a variable and a functor are implemented in the same way for Prop.

$\texttt{Prop-SPECAT}(f, i, \{j_1, \ldots, j_n\}, g) = f \wedge (i \leftrightarrow (j_1 \wedge \ldots \wedge j_n)).$
$\texttt{Prop-COMB}(f, i, \{j_1, \ldots, j_n\}, g) = f \wedge (i \leftrightarrow (j_1 \wedge \ldots \wedge j_n)).$

Unification of two variables: The operation ℜ-UNIF is a particular case of the ℜ-SPECAT operation.

$\texttt{Prop-UNIF}(f, i, j) = f \wedge (i \leftrightarrow j).$

Junction of abstract tuples: Let $f_1 \in \texttt{Prop}_p, f_2 \in \texttt{Prop}_q$.

$\texttt{Prop-JOIN}(f_1, f_2) = f_1 \wedge \texttt{denorm}\ [(p + 1), \ldots, (p + q)]\ f_2.$

4 Open Product

The purpose of this section is to propose a novel combination of domains achieving more precision than the direct product. This contribution is orthogonal to the Pat(ℜ) domain proposed in the previous section and the combination of the two ideas gives rise to an advanced methodological

and software support to build complex abstract domains. Probably the problem of combining abstract domains was first addressed by the Cousots. In [12, 13] they proposed the idea of a reduced domain whose underlying idea is to cluster together elements of the cartesian product which have the same concretization. As a consequence, the reduced product of a Galois insertion is itself a Galois insertion. The reduced product is a theoretical ideal but it does not suggest a particular implementation, since an implementation cannot use the concretization function. The main contribution of this section is the idea of open product, a new construct to combine abstract domains efficiently and accurately.

4.1 Overview

The problem we consider is to design a domain D from some given domains $D_1, \ldots, D_n$. The key idea behind the open product is the notion of open abstract interpretation. 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 abstract objects (e.g. properties not captured by the interpretation). The main benefit of open interpretations is the fact that abstract operations are able to receive information from the environment to improve their accuracy. Note that the idea of open interpretation is somewhat related to the idea of open semantics [3].

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 idea of open interpretations has many applications. It can be used to combine domains, as mentioned before. It can also be used to implement refinement operations which were proposed independently, but not formalized, in [6]. The motivation behind refinements comes from the fact that the open product only uses the initial abstract object for interaction. By letting the components interact after the operations, refinements enable to remove redundancies, i.e. they improve the components locally while not modifying the global concretization function. Refinements are special case of global operations. Finally, open interpretations can be used in the `Pat(`$\Re$`)` domain to let the $\Re$-domain use some structural information if necessary.

The notion of open product and open interpretation are both theoretical and practical tools. On the theory 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. As a consequence, these ideas provide advanced support to build sophisticated domains in a simple manner.

The rest of this section is organized as follows. In subsection 4.2, we define the concepts of query and open interpretation in whole generality. These notions allow to provide a consistent definition of open product, where the information contained by each component can be used by the other components through queries. Subsection 4.3 presents an example of application of this

approach to the case of the $\Re$-domain presented in Section 2, by showing the open product of `Prop` and of the abstract domains `OLIN` and `OPS` for linearity and sharing analysis. Subsection 4.4 further applies this approach to the interaction of the pattern and $\Re$-components in the open generic pattern domain `OPat`($\Re$). We conclude the section describing the open abstract interpretation on the domain `OMode` for mode analysis.

4.2 Semantics

4.2.1 Open Interpretations

Recall that a domain is a cpo with a partial order $\leq$ and an upper bound operation. We use *Bool* for the set $\{true, false\}$, $\leq_{TF}$ for the order `true` $\leq$ `false`, and $\leq_{FT}$ for the order `false` $\leq$ `true`.

Definition 12 (test) *A test is a boolean function $\mathcal{T} : Arg \rightarrow Bool$. For a given order $\leq_o$ on boolean values, the tests on the same set Arg can be partially ordered as follows:*

$$\mathcal{T} \leq \mathcal{T}' \Leftrightarrow \forall h \in Arg : \mathcal{T}(h) \leq_o \mathcal{T}'(h)$$

Definition 13 (query) *A query on the domain D is a pair $\mathcal{Q} = (\mathcal{P}, \leq)$, where $\leq$ is an ordering on boolean values and $\mathcal{P}$ is a monotone function which maps elements of the domain D onto tests.*

$$\mathcal{P} : D \rightarrow Arg_D \rightarrow Bool$$

For the sake of clarity, we often identify a query with its boolean function and use $\mathcal{Q}(d)$ instead of $\mathcal{P}(d)$.

Example On the domain of term tuples $\wp(ST_p)$ we can define the queries ($\mathcal{C}$-`GROUND`, $\leq_{TF}$), ($\mathcal{C}$-`MAYSHARE`,$\leq_{FT}$) and ($\mathcal{C}$-`LINEAR`, $\leq_{TF}$) for groundness sharing and linearity. The functions $\mathcal{C}$-`GROUND`: $ST_p \rightarrow I_p \rightarrow Bool$, $\mathcal{C}$-`MAYSHARE`: $ST_p \rightarrow I_p \times I_p \rightarrow Bool$, and $\mathcal{C}$-`LINEAR`: $ST_p \rightarrow I_p \rightarrow Bool$ are defined as follows.

$$\mathcal{C}\text{-}\texttt{GROUND}(S)(i) = \begin{cases} \texttt{true} & \text{if } \forall < t_1, \ldots, t_p > \in S : t_i \text{ is a ground term} \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$\mathcal{C}\text{-}\texttt{MAYSHARE}(S)(i,j) = \begin{cases} \texttt{false} & \text{if } \forall < t_1, \ldots, t_p > \in S : t_i, t_j \text{ do not share variables} \\ \texttt{true} & \text{otherwise} \end{cases}$$

$$\mathcal{C}\text{-}\texttt{LINEAR}(S)(i) = \begin{cases} \texttt{true} & \text{if } \forall < t_1, \ldots, t_p > \in S : t_i \text{ does not contain multiple} \\ & \text{occurrences of the same variable} \\ \texttt{false} & \text{otherwise} \end{cases}$$

where $S \subseteq ST_p$ is a set of p-tuples of terms.

Our notion of query interpretation is a slight generalization of the traditional notion of interpretation [10].

Definition 14 (query interpretation) *A query interpretation is a tuple* $\mathbf{D} = (D, \leq, \langle \mathtt{OP}_1, \ldots, \mathtt{OP}_n \rangle, \langle \mathcal{Q}_1, \ldots, \mathcal{Q}_m \rangle)$ *where:*

- D *is a domain;*
- $\leq$ *is the partial order on* D*;*
- $\mathtt{OP}_1, \ldots, \mathtt{OP}_n$ *are operations of signature* $\mathtt{OP}_i : D \to D$*;*[10]
- $\mathcal{Q}_1 \ldots, \mathcal{Q}_m$ *are queries on* D*.*

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 and open interpretation.

Definition 15 (open operation) *An open operation* $\mathtt{OP}$ *on the domain* D *is a function which maps a tuple of tests onto an operation. Let* $m \geq 0$*.*

$$\mathtt{OP} : (Arg \to Bool)^m \to D \to D$$

Observe that an operation $\mathtt{OP} : D \to D$ can be seen as a degenerate open operation which does not depend on tests. In the following, we assume without loss of generality that all operations use the same number of queries.

The notion of query interpretation is generalized according to the definition of open operations.

Definition 16 (open interpretation) *An open interpretation is a tuple* $\mathbf{D} = (D, \leq, \langle \mathtt{OP}_1, \ldots, \mathtt{OP}_n \rangle, \langle \mathcal{Q}_1, \ldots, \mathcal{Q}_m \rangle)$ *where:*

- D *is a domain;*
- $\leq$ *is the partial order on* D*;*
- $\mathtt{OP}_1, \ldots, \mathtt{OP}_n$ *are open operations on* D*;*
- $\mathcal{Q}_1 \ldots, \mathcal{Q}_m$ *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.

4.2.2 Open Abstract Interpretations

Now we consider the abstraction of query interpretations and of open interpretations.

Definition 17 (abstraction of queries) *Let* $\mathcal{Q}_1$ *and* $\mathcal{Q}_2$ *be two queries with the same Boolean order* $\leq$ *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 the following condition is satisfied.*

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

[10] For simplicity, we restrict our attention to unary operations. A generalization of the definition is straightforward.

Example In the case of Prop, we can consider queries for groundness, sharing, and linearity. The functions Prop-GROUND: $\texttt{Prop}_p \to I_p \to Bool$, Prop-MAYSHARE: $\texttt{Prop}_p \to I_p \times I_p \to Bool$, and Prop-LINEAR: $\texttt{Prop}_p \to I_p \to Bool$ are defined as follows. Let f be a formula of Prop.

$$\begin{array}{lcl} \texttt{Prop-GROUND}(f)(i) & \Leftrightarrow & (f \to i) \\ \texttt{Prop-MAYSHARE}(f)(i,j) & \Leftrightarrow & \neg(f \to i)\ \&\ \neg(f \to j) \\ \texttt{Prop-LINEAR}(f)(i) & \Leftrightarrow & (f \to i) \end{array}$$

It is not difficult to see that $(\texttt{Prop-GROUND}, \leq_{TF})$, $(\texttt{Prop-MAYSHARE}, \leq_{FT})$ and $(\texttt{Prop-LINEAR}, \leq_{TF})$ are abstractions of the concrete queries $(\mathcal{C}\texttt{-GROUND}, \leq_{TF})$, $(\mathcal{C}\texttt{-MAYSHARE}, \leq_{FT})$ and $(\mathcal{C}\texttt{-LINEAR}, \leq_{TF})$ introduced in the previous example.

Definition 18 (abstraction of query interpretations) *Consider two query interpretations* $\mathbf{D}_1 = (D_1, \leq_1, \langle \texttt{OP}^1_1, \ldots, \texttt{OP}^1_n \rangle, \langle \mathcal{Q}^1_1, \ldots \mathcal{Q}^1_m \rangle)$ *and* $\mathbf{D}_2 = (D_2, \leq_2, \langle \texttt{OP}^2_1, \ldots, \texttt{OP}^2_n \rangle, \langle \mathcal{Q}^2_1, \ldots, \mathcal{Q}^2_m \rangle)$.
We say that $\mathbf{D}_2$ *is an abstraction of* $\mathbf{D}_1$ *if:*

- D_2 *abstracts* D_1*;*
- *for* $1 \leq i \leq n$: $\texttt{OP}^2_i$ *abstracts* $\texttt{OP}^1_i$*, i.e.* $\forall d_2 \in D_2 :\ \texttt{OP}^1_i(Cc(d_2)) \leq_1 Cc(\texttt{OP}^2_i(d_2))$;
- *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 19 (open abstraction) *Consider a query interpretation* $\mathbf{D}_1 = (D_1, \leq_1, \langle \texttt{OP}^1_1, \ldots, \texttt{OP}^1_n \rangle, \langle \mathcal{Q}^1_1, \ldots \mathcal{Q}^1_m \rangle)$ *and an open interpretation* $\mathbf{D}_2 = (D_2, \leq_2, \langle \texttt{OP}^2_1, \ldots, \texttt{OP}^2_n \rangle, \langle \mathcal{Q}^2_1, \ldots, \mathcal{Q}^2_m \rangle)$.
We say that $\mathbf{D}_2$ *is an open abstraction of* $\mathbf{D}_1$ *if:*

- D_2 *abstracts* D_1*;*
- *for* $1 \leq i \leq n$: $\texttt{OP}^2_i$ *abstracts* $\texttt{OP}^1_i$*, i.e.*
  - $\texttt{OP}^2_i$ *is monotone with respect to the boolean orders on* $\mathcal{Q}^1_1, \ldots \mathcal{Q}^1_m$*;*
  - $\forall d_2 \in D_2\ \forall d_1 \in D_1 : d_1 \leq_1 Cc(d_2) \ \Rightarrow\ \texttt{OP}^1_i(d_1) \leq_1 Cc(\texttt{OP}^2_i(\langle \mathcal{Q}^1_1(d_1), \ldots, \mathcal{Q}^1_m(d_1) \rangle)(d_2))$.
- *for* $1 \leq i \leq m$: $\mathcal{Q}^2_i$ *abstracts* $\mathcal{Q}^1_i$*.*

4.2.3 Open Product

We are now in position to define the notion of open products of domains.

Definition 20 (open product) *Consider two open interpretations* $\mathbf{D}_1 = (D_1, \leq_1, \langle \texttt{OP}^1_1, \ldots, \texttt{OP}^1_n \rangle, \langle \mathcal{Q}^1_1, \ldots \mathcal{Q}^1_m \rangle)$ *and* $\mathbf{D}_2 = (D_2, \leq_2, \langle \texttt{OP}^2_1, \ldots, \texttt{OP}^2_n \rangle, \langle \mathcal{Q}^2_1, \ldots, \mathcal{Q}^2_m \rangle)$ *such that the order* $\leq_i$ *of* $\mathcal{Q}^1_i$ *is the same as that of* $\mathcal{Q}^2_i$*.*
The open product $\mathbf{D}_1 \otimes \mathbf{D}_2$ *is the query interpretation* $(D, \leq, \langle \texttt{OP}_1, \ldots, \texttt{OP}_n \rangle, \langle \mathcal{Q}_1, \ldots, \mathcal{Q}_m \rangle)$ *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* $\mathcal{Q}_i = (\mathcal{P}_i, \leq_i)$ *is defined by:*

  the order $\leq_i$ *is the same order as in* $\mathcal{Q}_i^1$ *and* $\mathcal{Q}_i^2$

  $\mathcal{P}_i = \mathcal{P}_i^1 \oplus \mathcal{P}_i^2$ *where*

  $$\oplus = \begin{cases} \wedge & \text{if the order } \leq_i \text{ is } \leq_{FT}; \\ \\ \vee & \text{if the order } \leq_i \text{ is } \leq_{TF}. \end{cases}$$

- *the operation* $\mathtt{OP}_i : (D_1 \times D_2) \rightarrow (D_1 \times D_2)$ *is defined by:*

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

Note that the operations use the best available queries by using the combination of queries from each domain.

Theorem 2 (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* $\mathbf{D}_0$*. Let* $\mathbf{D}_1$ *and* $\mathbf{D}_2$ *be open interpretations. Assume that both* $\mathbf{D}_1$ *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: Let $\mathbf{D}_0 = (D_0, \leq_0, \langle \mathtt{OP}_1^0, \ldots, \mathtt{OP}_n^0\rangle, \langle \mathcal{Q}_1^0, \ldots, \mathcal{Q}_m^0\rangle)$, $\mathbf{D}_1 = (D_1, \leq_1, \langle \mathtt{OP}_1^1, \ldots, \mathtt{OP}_n^1\rangle, \langle \mathcal{Q}_1^1, \ldots, \mathcal{Q}_m^1\rangle)$, and $\mathbf{D}_2 = (D_2, \leq_2, \langle \mathtt{OP}_1^2, \ldots, \mathtt{OP}_n^2\rangle, \langle \mathcal{Q}_1^2, \ldots, \mathcal{Q}_m^2\rangle)$. The three conditions of definition 18 hold.

1. **Domain:** Domain $D_1 \times D_2$ abstracts D_0 through the following concretization function:

   $$Cc : D_1 \times D_2 \rightarrow D_0$$
   $$Cc(d_1, d_2) = Cc(d_1) \sqcap Cc(d_2)$$

   This function is monotone by composition of monotone functions.

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

   $d_0 \leq_0 Cc(d_1), Cc(d_2)$
   $\Rightarrow\ \mathcal{Q}_i^0(d_0) \leq_i \mathcal{Q}_i^1(d_1), \mathcal{Q}_i^2(d_2)$
   $\Rightarrow\ \mathcal{Q}_i^0(d_0) \leq_i \mathcal{Q}_i(d_1, d_2)$

3. **Operations:** Operation $\mathtt{OP}_i$ abstracts $\mathtt{OP}_i^0$ since

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

   $\Rightarrow\ \mathtt{OP}_i^0(d_0) \leq_0 Cc(\mathtt{OP}_i^h(\langle \mathcal{Q}_1^0(d_0), \ldots, \mathcal{Q}_m^0(d_0)\rangle)(d_h))$
   (D_h is an open abstraction of $D_0, 1 \leq h \leq 2$)

   $\Rightarrow\ \mathtt{OP}_i^0(d_0) \leq_0 Cc(\mathtt{OP}_i^h(\langle \mathcal{Q}_1(d_1, d_2), \ldots, \mathcal{Q}_m(d_1, d_2)\rangle)(d_h))$
   ($\mathcal{Q}_j$ is an abstraction of $\mathcal{Q}_j^0, 1 \leq j \leq m, 1 \leq h \leq 2$)

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

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

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

■

Note that, in general, the existence of $\sqcap$ is trivially ensured, since $\mathbf{D}_0$ will be the concrete domain.

4.2.4 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 [6].

Specification 11 (refinement) *Let $\mathbf{D}_1$, $\mathbf{D}_2$ be respectively a query interpretation and an open interpretation such that $\mathbf{D}_2$ is an open abstraction of $\mathbf{D}_1$. An operation $\mathtt{REFINE}$: $(Arg_{D_1} \rightarrow Bool)^m \rightarrow D_2 \rightarrow D_2$ is a refinement operation of D_2 with respect to D_1 if for all $d_1 \in D_1$ and $d_2 \in D_2$, the following conditions are satisfied.*

- $d_1 \leq_1 Cc(d_2) \Rightarrow d_1 \leq_1 Cc(\mathtt{REFINE}(\langle \mathcal{Q}_1^1(d_1), \ldots, \mathcal{Q}_m^1(d_1)\rangle)(d_2))$;
- $\mathtt{REFINE}(\langle \mathcal{T}_1, \ldots, \mathcal{T}_m\rangle)(d_2) \leq_2 d_2$ *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 $\mathtt{REFINE}_1$ and $\mathtt{REFINE}_2$ are defined in $\mathbf{D}_1$ and $\mathbf{D}_2$ respectively. The corresponding operation $\mathtt{REFINE}$ on the open product is similar to traditional operations.

Definition 21 (refinement in the open product) *In the hypotheses and notation of theorem 2, assume that $\mathtt{REFINE}_1$ and $\mathtt{REFINE}_2$ are refinement functions for $\mathbf{D}_1$ and $\mathbf{D}_2$ with respect to $\mathbf{D}_0$. The refinement function $\mathtt{REFINE}$ on the open product $\mathbf{D}_1 \otimes \mathbf{D}_2$ is defined by:*

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

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

Definition 22 (refined abstract operation) *Under the hypotheses and notation of theorem 2, assume that $\mathtt{REFINE}_1$ and $\mathtt{REFINE}_2$ 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:*

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

It is easy to adapt the correctness proof to refined abstract operations. In general, the above definitions are sufficient to obtain a good accuracy. However, it may be the case that multiple applications of the refinement are useful. This leads to revisit the definition of the `REFINE` function in the open product.

Definition 23 (refinement in the open product revisited) *In the hypotheses and notation of theorem 2, assume that* $\texttt{REFINE}_1$ *and* $\texttt{REFINE}_2$ *are refinement functions for* $\mathbf{D}_1$ *and* $\mathbf{D}_2$ *with respect to* $\mathbf{D}_0$*. Define*

$$\begin{array}{lcl}(a_0, b_0) & = & (d_1, d_2) \\ (a_n, b_n) & = & (\texttt{REFINE}_1(\langle \mathcal{Q}_1(a_{n-1}, b_{n-1}), \ldots, \mathcal{Q}_m(a_{n-1}, b_{n-1})\rangle)(a_{n-1}) \ , \\ & & \ \texttt{REFINE}_2(\langle \mathcal{Q}_1(a_{n-1}, b_{n-1}), \ldots, \mathcal{Q}_m(a_{n-1}, b_{n-1})\rangle)(b_{n-1})\,) \qquad n > 0\end{array}$$

The operation `REFINE` *in the open product* $\mathbf{D}_1 \otimes \mathbf{D}_2$ *is now*

$$\texttt{REFINE}(d_1, d_2) = (a_m, b_m) \text{ for some } m \geq 0.$$

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.

4.3 Application I: $\Re$ as an Open Product

We have already outlined the generality of the framework for combining abstract domains presented above. In this section we show how to apply this approach to the multiple instantiation of the $\Re$-component of the domain `Pat(`$\Re$`)` developed in Section 3, combining the two contributions of this paper.

4.3.1 Overview

An $\Re$-domain is an abstraction of the concrete domain of term tuples $\wp(ST)$. We have already presented, in the examples of Section 4.2, the meaning of queries in this concrete domain and how an abstraction of these queries can be defined, for instance, in `Prop`. The main difference for the design of an $\Re$-domain is that the operations may be parameterized on tests. However, none of the specifications and implementations of Section 3 needs to be changed. Each $\Re$-operation on the domain `Pat(`$\Re$`)` is performed, as mentioned in Section 4.2.4, by a parallel execution of the open $\Re_i$-operations followed by a global refinement.

Summarizing, let $\Re = \Re_1 \otimes \ldots \otimes \Re_n$ be an open product of $\Re$-domains. In order to implement a cooperative interaction among the components, each domain $\Re_i$ is asked to specify:

i) *The tests called by* $\Re_i$, i.e. the queries with respect to whom the open operations on $\Re_i$ are abstraction of the corresponding concrete operations.

ii) *The queries supported by* $\Re_i$. In practice, a default value is automatically provided to the queries which are not explicitly supported by the domain $\Re_i$.

	($\mathcal{C}$-GROUND,$\leq_{TF}$)	($\mathcal{C}$-MAYSHARE,$\leq_{FT}$)	($\mathcal{C}$-LINEAR,$\leq_{TF}$)
OProp	yes	yes	yes
OPS	not	yes	not
OLIN	not	not	yes
default abstr.	(false, $\leq_{TF}$)	(true, $\leq_{FT}$)	(false, $\leq_{TF}$)

Table 1: Queries abstracted by the $\Re$-domains

	GROUND	MAYSHARE	LINEAR
OProp	not	not	not
OPS	yes	not	yes
OLIN	yes	yes	yes

Table 2: Tests called by $\Re$-domains

iii) *A refinement function* $\Re_i$-REFINE. This open abstract operation uses the available information from the other components to refine the precision of the abstract term tuple. The default implementation of $\Re_i$-REFINE is the identity function on $\Re_i$.

Let us apply the definitions presented in Section 4.2 to obtain a combined analysis of groundness, linearity, and sharing. At this end, we consider three open interpretations: OProp for groundness, OPS for sharing and OLIN for linearity.

We need to embed Prop, presented in section 3.4, in an open interpretation OProp, and to specify the two open interpretations OPS and OLIN. Their open product will be fully specified in Section 4.3.5. Table 1 illustrates the queries supported in a non-trivial way by the $\Re$-domains considered and shows the default abstraction of each of them. Table 2 summarizes the tests actually used by these $\Re$-domains.

4.3.2 The Open Interpretation OProp

We choose to represent Prop as a query interpretation OProp, since Prop is generally very accurate and would not benefit much from other domains.[11] The *operations* in OProp are $\langle$OProp-REN, ..., OProp-JOIN$\rangle$, and the *queries* are $\langle$(OProp-GROUND, $\leq_{TF}$), (OProp-MAYSHARE, $\leq_{FT}$), (OProp-LINEAR, $\leq_{TF}$)$\rangle$. The *refinement operation* for OProp is provided by default: OProp-REFINE$(f) = f$.

4.3.3 The Open Interpretation OPS

Domain The abstract domain OPS (inspired by the sharing component described in [24]) specifies the possible pair-sharing of variables between terms. As usual, let p be an integer. The elements of $\texttt{OPS}_p$ are binary and symmetrical relations $ps : I_p \times I_p$. We denote by OPS the union of all the $\texttt{OPS}_p$ for p integer. The ordering between two abstract elements ps_1, ps_2 over the same set of variables is defined as follows: $ps_1 \leq ps_2$ if $\forall(i,j) : ps_1(i,j) \Rightarrow ps_2(i,j)$. The element ps specifies that the

[11] Of course, other choices are possible in other contexts.

terms can only share variables when the ps relation allows it.

$$Cc : \texttt{OPS}_p \rightarrow \wp(ST_p)$$
$$Cc(ps) = \{< t_1, \ldots, t_p > \mid \forall i, j \in I_p : \ var(t_i) \cap var(t_j) \neq \emptyset \Rightarrow ps(i,j)\}.$$

Queries `OPS` supports non-trivially the query ($\texttt{OPS-MAYSHARE}, \leq_{FT}$) as follows:

$$\texttt{OPS-MAYSHARE}(ps)(i,j) \ \Leftrightarrow \ (i,j) \in ps.$$

The other queries defined on the concrete domain $\wp(ST_p)$ are supported by `OPS` through their default implementation.

Operations It remains to specify the abstract operations on the abstract domain `OPS`. Let $\tilde{ps}$ denote the symmetrical relation deduced from ps. The operations `OPS-REN`, `OPS-INTR`, `OPS-PROJ`, `OPS-UNION` and `OPS-JOIN` do not depend on tests. On the other hand, the operations `OPS-SPECAT`, `OPS-COMB` and `OPS-UNIF` are properly open operations.

Renaming: Let $p' \geq p$ and let $t : I_{p'} \rightarrow I_{p'}$ be a renaming of indices, i.e. a bijective function from the set $\{1, \ldots, p'\}$ to itself.

$\texttt{OPS-REN}()(ps, t) = \{(t(i), t(j)) \mid ps(i,j)\}.$

Introduction of new variables: Let $t(i) = i$ for $1 \leq i \leq m$ and $t(i) = (i-k)$ for $m+k < i \leq p+k$.

$\texttt{OPS-INTR}()(ps, m, k) = \{(i,j) \mid ps(t(i), t(j))\} \cup \{(i,i) \mid m < i \leq m+k\}.$

Projection of a variable:

$\texttt{OPS-PROJ}()(ps, j) = \{(h,k) \mid ps(h,k) \,\&\, h \neq j \,\&\, k \neq j\}$

Union of abstract tuples:

$\texttt{OPS-UNION}()(ps_1, ps_2) = \{(i,j) \mid ps_1(i,j) \vee ps_2(i,j)\}.$

Unification of a variable and a functor: The two specializations of the unification of a variable and a functor, are implemented as follows. `OPS-SPECAT` is an open abstraction of $\mathcal{C}$`-SPECAT` with respect to the tests ($\mathcal{C}$`-GROUND`$, \leq_{TF}$) and ($\mathcal{C}$`-LINEAR`$, \leq_{TF}$). The intuition behind the implementation is as follows. When the term t_i is ground, no sharing has to be propagated to the (new variables) $t_{j_1}, \ldots, t_{j_n}$, which surely become ground after this operation. Moreover, if t_i is linear, then the terms $t_{j_1}, \ldots, t_{j_n}$ remain independent.

$\texttt{OPS-SPECAT}(\langle \texttt{GROUND}, \texttt{LINEAR} \rangle)(ps, i, \{j_1, \ldots, j_n\}, g) = ps'$

$$ps' = \begin{cases} ps & \text{if } \texttt{GROUND}(i) \\ ps \cup \tilde{ps_0} \cup \tilde{ps_1} \cup ps_2 & \text{otherwise} \end{cases}$$

where:

$ps_0 = \{(i, j_h) \mid 1 \leq h \leq n\}.$

$ps_1 = \{(k, j_h) \mid 1 \leq h \leq n \;\&\; ps(i,k) \;\&\; k \notin \{j_1, \ldots, j_n\}\}.$

$$ps_2 = \begin{cases} \emptyset & \text{if } \texttt{LINEAR}(i) \\ \{(j_h, j_k) \mid 1 \leq h, k \leq n\} & \text{otherwise} \end{cases}$$

The open operation `OPS-COMB` is an open abstraction of $\mathcal{C}$-`COMB` with respect to ($\mathcal{C}$-`GROUND`,$\leq_{TF}$).

`OPS-COMB(GROUND)`$(ps, i, \{j_1, \ldots, j_n\}, g) = ps'$

$$ps' = \begin{cases} ps & \text{if } \texttt{GROUND}(j_1) \& \ldots \& \texttt{GROUND}(j_n) \\ ps \cup \tilde{ps}_1 \cup \tilde{ps}_2 & \text{otherwise} \end{cases}$$

$$\begin{aligned} ps_1 &= \{(i, j_h) \mid 1 \leq h \leq n\} \\ ps_2 &= \{(i,k) \mid \exists h : 1 \leq h \leq n : ps(j_h, k)\}. \end{aligned}$$

Unification of two variables: The open operation `OPS-UNIF` is an open abstraction of $\mathcal{C}$-`UNIF` with respect to the query ($\mathcal{C}$-`GROUND`,$\leq_{TF}$).

`OPS-UNIF(GROUND)`$(ps, i, j) = \tilde{ps}'$, where

$$ps' = \begin{cases} ps & \text{if } \texttt{GROUND}(i) \vee \texttt{GROUND}(j) \\ ps \cup \{(i,j)\} \cup & \\ \{(k,l) \mid \exists k', l' : ps(k', k) \,\&\, ps(l', l) \,\&\, k', l' \in \{i, j\}\} \cup & \\ \{(i,k) \mid ps(j,k)\} \cup \{(j,k) \mid ps(i,k)\} & \text{otherwise.} \end{cases}$$

Junction of two abstract tuples: Let $ps_1, ps_2 \in$ `OPS`, and let I_{p_1} and I_{p_2} the sets of indices associated to ps_1, ps_2, respectively.

`OPS`$-$`JOIN`$()(ps_1, ps_2) = ps_1 \cup \{(i + p_1, j + p_1) \mid ps_2(i,j)\}.$

Refinement The following refinement function can be applied to $ps \in$ `OPS` in the presence of groundness information. Let $W = \{i \mid$ `GROUND`$(i)\}$.

`OPS-REFINE(GROUND)`$(ps) = \{(i,j) \mid ps(i,j) \,\&\, i \notin W \,\&\, j \notin W\}.$

4.3.4 The Open Interpretation `OLIN`

Now we introduce a simple domain for linearity. Even though more complicated domains for linearity can be defined, the domain `OLIN` shows how a simple domain heavily benefits from queries when combined with domains providing groundness and sharing information.

Domain The elements of `OLIN`$_p$ are sets of indices $lin \subseteq I_p$. The order is defined as follows: if $lin_1, lin_2 \in$ `OLIN`$_p$, we say that $lin_1 \leq lin_2$ if $lin_1 \supseteq lin_2$.

The meaning of a set lin is given by the concretization function that specifies that the set represents all the $p-$tuples of terms that satisfy the linearity constraint.

$$Cc : \texttt{OLIN}_p \to \wp(ST_p)$$
$$Cc(lin) = \{< t_1, \ldots, t_p > \mid \text{ if } i \in lin \text{ then } t_i \text{ is linear } \}$$

Queries The domain OLIN supports the query OLIN-LINEAR by

$$\texttt{OLIN-LINEAR}(lin)(i) \Leftrightarrow i \in lin.$$

The other queries on $\wp(ST_p)$ are not explicitly abstracted in OLIN. However, they can be seen supported through their default approximations.

Operations It remains now to specify the abstract operations on the abstract domain OLIN. The operations OLIN-REN, OLIN-INTR, OLIN-PROJ, OLIN-UNION and OLIN-JOIN do not depend on tests. On the other hand, the operations OLIN-SPECAT, OLIN-COMB and OLIN-UNIF are properly open operations.

Renaming of an abstract tuple: Given a renaming of indices r, the new set is obtained by simultaneously replacing each occurrence of i by $r(i)$.

$\texttt{OLIN-REN}(\,)(lin, r) = \{r(i) \mid i \in lin\}$.

Introduction of new variables: Let $t(i) = i$ for $1 \leq i \leq m$ and $t(i) = (i + k)$ for $m < i \leq p$.

$\texttt{OLIN-INTR}(\,)(lin, m, k) = \{t(i) \mid i \in lin\} \cup \{i \mid m < i \leq m + k\}$

Projection of a variable: Projection is simply defined as:

$\texttt{OLIN-PROJ}(\,)(lin, j) = lin \setminus \{j\}$.

Union of abstract tuples: The UNION of two sets lin_1, lin_2 corresponds to their intersection:

$\texttt{OLIN-UNION}(\,)(lin_1, lin_2) = lin_1 \cap lin_2$.

Unification of a variable and a functor: OLIN-SPECAT is an abstraction of $\mathcal{C}$-SPECAT with respect to the query ($\mathcal{C}$-LINEAR,$\leq_{TF}$). The operation $\texttt{OLIN-SPECAT}(\texttt{LINEAR})(lin, i, \{j_1, \ldots, j_n\}, g)$ is implemented by removing from lin the indices which correspond to terms possibly nonlinear because of this unification. Recall that $\{t_{j_1}, \ldots, t_{j_n}\}$ are free variables. If t_i is non-linear then one of $\{t_{j_1}, \ldots, t_{j_n}\}$ may become non-linear. Observe that if the term t_i is ground, it is linear, and also $t_{j_1}, \ldots, t_{j_n}$, which become ground after the unification step, will remain linear.

$\texttt{OLIN-SPECAT}(\texttt{LINEAR})(lin, i, \{j_1, \ldots, j_n\}, g) = lin'$

$$lin' = \begin{cases} lin & \text{if } \texttt{LINEAR}(i) \\ lin \setminus \{j_1, \ldots, j_n\} & \text{otherwise} \end{cases}$$

The open operation OLIN-COMB is an abstraction of $\mathcal{C}$-COMB with respect to the tuple of queries $\langle(\mathcal{C}\text{-LINEAR},\leq_{TF}), (\mathcal{C}\text{-MAYSHARE}, \leq_{FT})\rangle$.

$\texttt{OLIN-COMB}(\langle\texttt{LINEAR}, \texttt{MAYSHARE}\rangle)(lin, i, \{j_1, \ldots, j_n\}, g) = lin'$

$$lin' = \begin{cases} lin \setminus W & \text{if } (\exists k : \neg\texttt{LINEAR}(j_k)) \vee (\exists h,k : \texttt{MAYSHARE}(j_h, j_k)) \\ lin & \text{otherwise} \end{cases}$$

$$W = \{k \in lin \mid \texttt{MAYSHARE}(i,k)\}$$

Unification of two variables: The operation `OLIN-UNIF` is an abstraction of $\mathcal{C}$-`UNIF` with respect to the tuple of queries $\langle(\mathcal{C}\text{-}\texttt{GROUND},\leq_{TF}),(\mathcal{C}\text{-}\texttt{LINEAR},\leq_{TF}),(\mathcal{C}\text{-}\texttt{MAYSHARE},\leq_{FT})\rangle$.

$$\texttt{OLIN-UNIF}(\langle\texttt{GROUND},\texttt{LINEAR},\texttt{MAYSHARE}\rangle)(lin,i,j) = lin'$$

$$lin' = \begin{cases} lin & \text{if } \texttt{GROUND}(i) \vee \texttt{GROUND}(j) \vee (\texttt{LINEAR}(i) \wedge \texttt{LINEAR}(j) \wedge \neg\texttt{MAYSHARE}(i,j)) \\ lin \setminus W & \text{otherwise} \end{cases}$$

$$W = \{k \in lin \mid \texttt{MAYSHARE}(i,k) \vee \texttt{MAYSHARE}(j,k)\}$$

Junction of abstract tuples: Let $lin_1 \in \texttt{OLIN}_{p_1}$ and $lin_2 \in \texttt{OLIN}_{p_2}$.

$$\texttt{OLIN-JOIN}(\,)(lin_1, lin_2) = lin_1 \cup \{(i + p_1) \mid i \in lin_2\}.$$

Refinement The refinement function adds to lin the indices representing ground terms.

$$\texttt{OLIN-REFINE}(\texttt{GROUND})(lin) \;=\; lin \cup \{i \mid \texttt{GROUND}(i)\}.$$

4.3.5 The open product $\Re = \texttt{OProp} \otimes \texttt{OLIN} \otimes \texttt{OPS}$.

We are ready to fully specify the open product $\Re = \texttt{OProp} \otimes \texttt{OLIN} \otimes \texttt{OPS}$, of the three open abstract interpretations introduced so far. The components of $\Re = (R, \leq, \Re\text{-}\texttt{REFINE}, \langle\Re\text{-}\texttt{REN},\ldots,\Re\text{-}\texttt{JOIN}\rangle, \langle(\Re\text{-}\texttt{GROUND},\leq_{TF}),(\Re\text{-}\texttt{MAYSHARE},\leq_{FT}),(\Re\text{-}\texttt{LINEAR},\leq_{TF})\rangle)$ are as follows.

- R is the cartesian product of the three domains.
- The partial order $\leq$ is $(\rightarrow, \supseteq, \subseteq)$.
- The queries of the open product are defined by:

$$\begin{cases} \Re\text{-}\texttt{GROUND}(f,lin,ps) &=& \texttt{Prop-GROUND}(f) \\ \Re\text{-}\texttt{MAYSHARE}(f,lin,ps) &=& \texttt{Prop-MAYSHARE}(f) \wedge \texttt{OPS-MAYSHARE}(ps) \\ \Re\text{-}\texttt{LINEAR}(f,lin,ps) &=& \texttt{Prop-LINEAR}(f) \vee \texttt{OLIN-LINEAR}(lin) \end{cases}$$

- The operation $\Re$-`REFINE` is obtained, according to Definition 23, as follows:

$$\Re\text{-}\texttt{REFINE}(f,lin,ps) = (f',lin',ps')$$

$$\begin{cases} f' &=& \texttt{OProp-REFINE}(f) \\ lin' &=& \texttt{OLIN-REFINE}(\Re\text{-}\texttt{GROUND}(f,lin,ps))(lin) \\ ps' &=& \texttt{OPS-REFINE}(\Re\text{-}\texttt{GROUND}(f,lin,ps))(ps) \end{cases}$$

Observe that no further iteration is necessary in this case.

Now we can define the other operations on the open product. They may be split in two sets.

The operations $\Re$-REN, $\Re$-INTR, $\Re$-PROJ, $\Re$-UNION and $\Re$-JOIN are obtained by refining the pointwise combination of the corresponding operations in each domain, since none of them is strictly open. For instance,

$$\Re\text{-UNION}((f_1, lin_1, ps_1), (f_2, lin_2, ps_2)) = \Re\text{-REFINE}(f', lin', ps')$$

$$\begin{cases} f' & = & \text{OProp-UNION}(f_1, f_2) \\ lin' & = & \text{OLIN-UNION}(\,)(lin_1, lin_2) \\ ps' & = & \text{OPS-UNION}(\,)(ps_1, ps_2). \end{cases}$$

Observe that in some cases the refinement is redundant and can be skipped.

The operations $\Re$-SPECAT, $\Re$-COMB and $\Re$-UNIF are more interesting, since they are open operations. For instance,

$$\Re\text{-SPECAT}((f, lin, ps), i, \{j_1, \ldots, j_n\}, g) = \Re\text{-REFINE}(f', lin', ps')$$

$$\begin{cases} f' & = & \text{OProp-SPECAT}(f, i, \{j_1, \ldots, j_n\}, g) \\ lin' & = & \text{OLIN-SPECAT}(\Re\text{-LINEAR}(f, lin, ps))(lin, i, \{j_1, \ldots, j_n\}, g) \\ ps' & = & \text{OPS}-\text{SPECAT}(\langle \Re\text{-GROUND}(f, lin, ps), \Re\text{-LINEAR}(f, lin, ps) \rangle)(ps, i, \{j_1, \ldots, j_n\}, g) \end{cases}$$

The operations $\Re$-COMB and $\Re$-UNIF are defined similarly.

4.4 Application II: The Open Generic Pattern Domain OPat{$\Re$}

4.4.1 Overview

In the previous section, we have shown how the $\Re$-domain can be built as an open product of several open abstract interpretations. In this section, we go one step further and show that the $\Re$-domain itself can be viewed as an open interpretation inheriting automatically the information maintained by the pattern component. Although providing structural information is not useful for most abstract interpretations such as OProp, there exist some interpretations where it can improve accuracy significantly. A typical example is the domain OMode which maintain information on the mode of each subterm. For instance, OMode does not maintain groundness dependencies between subterms (contrary to OProp) and hence may results in significant loss of accuracy. The availability of structural information reduces this loss of accuracy. The main contribution of this subsection is to define a new generic domain OPat($\Re$) which can be viewed informally as $SV \times (FRM \otimes \Re)$. None of the definitions needs to be changed since the frm component will not use the information in $\Re$ but we need to specify its queries.

4.4.2 Queries

The pattern component supports queries on the structure of terms. These queries specify whether a term is structured or not, and, if it is the case, which is the arity and which are the main subterms. It is clear that all these properties can be easily expressed by means of boolean functions ordered by

$\leq_{TF}$. For practical reasons, we will use an equivalent representation in terms of partial functions. The concrete specification is:

$$\mathcal{C}\text{-}\texttt{STRUCTURED}(S)(i) = \begin{cases} \texttt{true} & \text{if } \forall < t_1, \ldots, t_p > \in S : t_i \text{ is a structured term} \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$\mathcal{C}\text{-}\texttt{ARITY}(S)(i) = \begin{cases} n & \text{if } \forall < t_1, \ldots, t_p > \in S : t_i = f(t_{j_1}, \ldots, t_{j_n}) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{C}\text{-}\texttt{ARGUMENT}(S)(i,h) = \begin{cases} i_h & \text{if } \forall < t_1, \ldots, t_p > \in S : t_i = f(t_{i_1}, .., t_{i_h}, .., t_{i_n}) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

where S is a set of concrete tuples in ST.

Each domain FRM_p supports the functions above as follows.

$$FRM\text{-}\texttt{STRUCTURED}(frm)(i) = \begin{cases} \texttt{true} & \text{if } frm(i) = f(j_1, \ldots, j_n) \\ \texttt{false} & \text{if } frm(i) = undef \end{cases}$$

$$FRM\text{-}\texttt{ARITY}(frm)(i) = \begin{cases} n & if\ frm(i) = f(j_1, \ldots, j_n) \\ \textit{undefined} & \text{if } frm(i) = undef. \end{cases}$$

$$FRM\text{-}\texttt{ARGUMENT}(frm)(i,h) = \begin{cases} i_h & \text{if } FRM_p\text{-}\texttt{ARITY}(frm)(i) = n \ \& \\ & h \leq n \ \& \ frm(i) = f(i_1, ..i_h .., i_n) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

In the rest of this section we introduce the open interpretation on the domain `OMode`, whose operations heavily depend on the pattern information provided by the frm component. The definitions of the operations in `OPat(MODE)` follow the lines of what we have seen in Section 4.3.5. Analogously, we can combine `OMode` with other domains obtaining, for instance, `OPat(OMode` $\otimes$ `OPS)`.

4.4.3 An Auxiliary Domain of Modes

Let us revisit the domain of modes defined in [33]. This domain will be the basis of the abstract domain `OMode`.

The modes are taken from the set $Modes = \{\bot, ground, var, ngv, novar, gv, noground, any\}$ and satisfy the ordering relationship defined by the diagram depicted in Figure 1, where the nodes are modes and an arc between nodes i_1 and i_2 with i_1 above i_2 means that $i_1 > i_2$. The ordering relation between modes is simply the transitive closure of the ordering implied by the above diagram. The semantics of the modes can be given by the following concretization functions:

$$\begin{array}{lcl} Cc(ground) & = & \{t \mid t \text{ is a ground term }\}; \\ Cc(var) & = & \{t \mid t \text{ is a variable }\}; \\ Cc(ngv) & = & \{t \mid t \text{ is not a variable nor a ground term }\}; \\ Cc(lub(M_1, M_2)) & = & Cc(M_1) \cup Cc(M_2). \end{array}$$

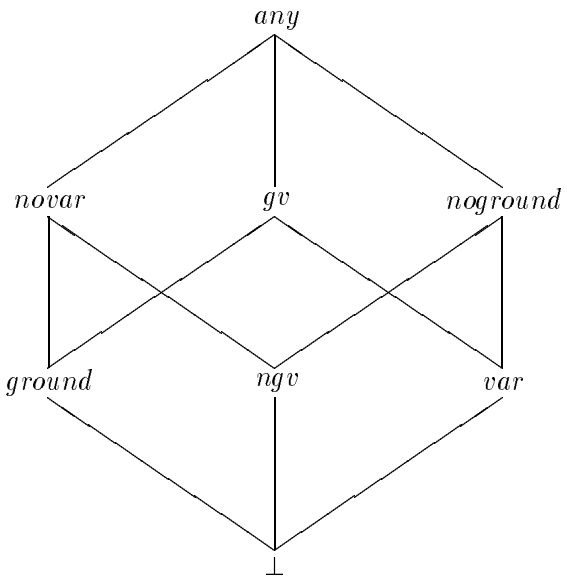

Figure 1: The Ordering of Modes as an Hasse diagram

We now turn to the various operations on modes.

$\texttt{CONS}(n, M_1, \ldots, M_n) = M'$

This operation, called Construction of Modes, will be used to compute the mode of a complex term $f(t_1, \ldots, t_n)$ from the modes of the subterms $t_1, \ldots, t_n$. For instance, if all the modes M_i are ground, it is clear that the result is ground. On the other hand, if one of the M_i is $noground$, the resulting term is ngv. It should be useful for a type domain including lists, for example. We keep it in order to preserve genericity of the presentation.

Specification 12 *Let $M_1, \ldots, M_n$ be modes, $t_1, \ldots, t_n$ be terms, and f be any n-ary function symbol. The operation satisfies the following property:*

$$\forall i : 1 \leq i \leq n : t_i \in Cc(M_i) \Rightarrow f(t_1, \ldots, t_n) \in Cc(M').$$

Implementation 13 *Denote by C the condition $\exists i : 1 \leq i \leq n : M_i = \bot$. Then the operation can be implemented as follows:*

$$\begin{array}{rcll} T' & = & \bot & \textit{if } C; \\ & & ground & \textit{if } \forall i : 1 \leq i \leq n : M_i = ground; \\ & & ngv & \textit{if } \neg C \textit{ and } \exists i : 1 \leq i \leq n : M_i \leq noground; \\ & & novar & \textit{otherwise.} \end{array}$$

$\texttt{EXTR}(M, n) = (M'_1, \ldots, M'_n)$

This operation, called Extraction of Modes, is the reverse of the previous operation. It computes the best possible modes of terms $t_1, \ldots, t_n$ such that for any-ary functor f, $f(t_1, \ldots, t_n)$

has mode M. For instance, if $f(t_1, \ldots, t_n)$ is ground, it is clear that all its arguments are ground as well. This operation is used when a term with an undefined pattern is unified with a term whose principal functor is known to be an n-ary functor f. (See operation `OMode-SPECAT`.)

Specification 13 *Let M be a mode, f be any function symbol of arity n, and $t_1, \ldots, t_n$ being terms. The operation satisfies the following property:*

$$f(t_1, \ldots, t_n) \in Cc(M) \;\Rightarrow\; \forall i : 1 \leq i \leq n : t_i \in Cc(M_i').$$

Implementation 14 *The operation can be implemented as follows:*

$$\begin{array}{lcll} M_i' & = & \bot & \text{if } M \in \{\bot, var\}; \\ & = & ground & \text{if } M \in \{ground, gv\}; \\ & = & noground & \text{if } M \in \{noground, ngv\} \text{ and } n = 1; \\ & = & any & \text{otherwise.} \end{array}$$

$\texttt{RECOM}(M, M_1, \ldots, M_n) = M'$

This operation, called Recomputing of Modes, is used during abstract unification to recompute the mode of a compound term when some of its subterms may have been instantiated (resulting in new subterms $t_1, \ldots, t_n$, with modes $M_1, \ldots, M_n$).

Specification 14 *Let $M, M_1, \ldots, M_n$ be modes, $t, t_1, \ldots, t_n$ be terms and f any n-ary function symbol. The operation satisfies the following property:*

$$t \in Cc(M) \,\&\, \forall i : 1 \leq i \leq n : t_i \in Cc(M_i) \,\&\, \exists \sigma : t\sigma = f(t_1, \ldots, t_n) \;\Rightarrow\; f(t_1, \ldots, t_n) \in Cc(M').$$

Implementation 15 *The implementation is $M' = lub(M_1', M_2')$ where*

$$\begin{array}{lcll} M_1' & = & \texttt{CONS}(n, M_1, \ldots, M_n) & \text{if } M \neq \bot \text{ and } M \neq ground; \\ & = & \bot & \text{otherwise.} \\ M_2' & = & ground & \text{if } M \geq ground \text{ and } \forall i : M_i \geq ground; \\ & = & \bot & \text{otherwise.} \end{array}$$

$\texttt{UAT}(M_1, M_2) = M'$

Given the modes of two terms, this operation, called Abstract Unification of Modes, returns the mode resulting of the unification of the terms.

Specification 15 *Let M_1, M_2 be modes, t_1, t_2 be terms and σ be a substitution. The operation satisfies the following property:*

$$t_1 \in Cc(M_1) \,\&\, t_2 \in Cc(M_2) \,\&\, \sigma \text{ is a mgu of } t_1, t_2 \;\Rightarrow\; t_1\sigma \in Cc(M').$$

The implementation is depicted in Table 3.

	bottom	var	ngv	ground	noground	gv	novar	any
bottom	bottom	bottom	bottom	bottom	bottom	bottom	bottom	bottom
var	bottom	var	ngv	ground	noground	gv	novar	any
ngv	bottom	ngv	novar	ground	novar	novar	novar	novar
ground	bottom	ground	ground	ground	ground	ground	ground	ground
noground	bottom	noground	novar	ground	any	any	novar	any
gv	bottom	gv	novar	ground	any	gv	novar	any
novar	bottom	novar	novar	ground	novar	novar	novar	novar
any	bottom	any	novar	ground	any	any	novar	any

Table 3: Abstract Unification of Modes

M	bottom	var	ngv	ground	noground	gv	novar	any
M'	bottom	any	novar	ground	any	any	novar	any

Table 4: Abstract Instantiation of Modes

$\texttt{IAT}(M) = M'$

This operation, called Abstract Instantiation of Modes, computes the mode M' of a term whose mode is M and which has been arbitrarily instantiated.

Specification 16 *Let M be a mode. For any term t and substitution σ, the following holds:*

$$t \in Cc(M) \Rightarrow t\sigma \in Cc(M').$$

The implementation is depicted in Table 4.

$\texttt{IAT}_2(M_1, M_2) = M'$

This operation, called Specialized Abstract Instantiation, computes the mode M' of a term whose mode is M_1 and which has been instantiated by a substitution of a variable to a term whose mode is M_2.

Specification 17 *Let M_1, M_2 be modes, t_1, t_2 be terms and y be a variable. The operation satisfies the following property:*

$$t_1 \in Cc(M_1)\ \&\ t_2 \in Cc(M_2)\ \Rightarrow\ t_1\{y \leftarrow t_2\} \in Cc(M').$$

Once again the implementation can be given in the form of a table shown in Table 5. The values of the first argument are given vertically.

4.4.4 The Open Pattern Domain OPat{OMode}

Domain As usual, let I_p be the set of indices $\{1, \ldots, p\}$. The elements of $\texttt{OMode}_p$ are the functions $mo : I_p \rightarrow Modes_\perp$ that associates a mode with each index in I_p where $Modes_\perp = Modes \setminus \{\perp\}$.

	bottom	var	ngv	ground	noground	gv	novar	any
bottom	bottom	bottom	bottom	bottom	bottom	bottom	bottom	bottom
var	bottom	var	noground	gv	noground	gv	any	any
ngv	bottom	ngv	ngv	novar	ngv	novar	novar	novar
ground	bottom	ground	ground	ground	ground	ground	ground	ground
noground	bottom	noground	noground	any	noground	any	any	any
gv	bottom	gv	any	gv	any	gv	any	any
novar	bottom	novar	novar	novar	novar	novar	novar	novar
any	bottom	any	any	any	any	any	any	any

Table 5: Specialized Abstract Instantiation of Modes: the values of the first argument are given vertically.

The meaning of a function mo is given in terms of a concretization function that specifies that mo represents all tuples of terms that satisfy the modes componentwise.

$$Cc : \texttt{OMode}_p \rightarrow \wp(ST_p)$$
$$Cc(mo) \;=\; \{< t_1, \ldots, t_p > \mid t_i \in Cc(mo(i)) \text{ for all } 1 \leq i \leq p\}$$

The domain $\texttt{OMode}_p$ is ordered componentwise, as follows: $mo_1 \leq mo_2$ iff $\forall i \in I_p : mo_1(i) \leq mo_2(i)$. In the following, we denote by $\texttt{OMode}$ the union of all $\texttt{OMode}_p$ $(p \geq 0)$.

Queries $\texttt{OMode}$ supports the queries $(\texttt{OMode-GROUND}, \leq_{TF})$ and $(\texttt{OMode-LINEAR}, \leq_{TF})$, as follows.

$$\begin{array}{lcl} \texttt{OMode-GROUND}(mo)(i) & \Leftrightarrow & mo(i) = ground \\ \texttt{OMode-LINEAR}(mo)(i) & \Leftrightarrow & (mo(i) = ground) \;\vee\; (mo(i) = var) \;\vee\; (mo(i) = gv) \end{array}$$

The abstraction of other queries, like $(\mathcal{C}\text{-}\texttt{MAYSHARE}, \leq_{FT})$, is by default.

Operations

Renaming: Let $p' \geq p$ and let $r : I_{p'} \rightarrow I_{p'}$ be a renaming of indices, i.e. a bijective function from the set $\{1, \ldots, p'\}$ to itself.

$\texttt{OMode-REN}()(mo, r) = mo'$

$mo'(r(i)) = mo(i)$.

Introduction of new variables:

$\texttt{OMode-INTR}()(mo, m, k) = mo'$, where

$$mo'(i) \;=\; \begin{cases} mo(i) & 1 \leq i \leq m \\ var & m < i \leq m + k \\ mo(i-k) & m + k < i \leq p + k. \end{cases}$$

Projection of a variable:

$\texttt{OMode-PROJ}()(mo, j) = mo'$

$$mo'(i) = \begin{cases} mo(i) & 1 \leq i < j \\ mo(i+1) & j \leq i \leq p-1. \end{cases}$$

Union of abstract tuples:

$\texttt{OMode-UNION}()(mo_1, mo_2) = mo'$, where

$mo'(i) = \texttt{LUB}(mo_1(i), mo_2(i))$.

Unification of a variable and a functor: The open operation `OMode-SPECAT` is probably the most complicated operation. Note that, since in general nothing is known about the pattern of t_i, the operation has to take into account all possible cases compatible with its mode.
Recall that this operation modifies the state mo by adding information about the unification of the term t_i and the structured term $f(t_{j_1}, \ldots, t_{j_n})$, where $t_{j_1}, \ldots, t_{j_n}$ are new distinct variables not occurring in the other terms. Let I_p be the set of indices associated to ℓ and let $i, j_1, \ldots j_n \leq p$. Assume, for simplicity, that the indices $\{j_1, \ldots, j_n\}$ are exactly $\{(p-n+1), \ldots, p\}$. Let $(M_1, \ldots, M_n) = \texttt{EXTR}(mo(i), n)$, and let $S = \langle$`MAYSHARE, STRUCTURED, ARITY, ARGUMENT`$\rangle$.

$\texttt{OMode-SPECAT}(S)(mo, i, \{j_1, \ldots, j_n\}, f) = mo'$, where $mo'(k)$ is as follows.

1. $1 \leq k \leq p - n$

if $\neg\texttt{MAYSHARE}(i, k)$
then $mo'(k) = mo(k)$

else if $k = i$
then
let $M = \texttt{RECOM}(mo(i), M_1, \ldots, M_n)$ **in**
$mo'(k) = \texttt{LUB}(M, \texttt{CONS}(n, var, \ldots, var))$ **if** $mo(i) \geq var$
$mo'(k) = M$ **otherwise**

else if ($k \neq i$ **and** $\texttt{STRUCTURED}(k)$
and $\texttt{ARITY}(k) = n$
and $\texttt{ARGUMENT}(k, h) = k_h$ `for` $h \in [1..n]$)
then $mo'(k) = \texttt{RECOM}(mo(k), mo'(k_1), \ldots, mo'(k_m))$

else if $k \neq i$ `and (STRUCTURED(`k`)= false)`
then
$mo'(k) = \texttt{IAT}_2(mo(k), \texttt{CONS}(n, var, \ldots, var))$ **if** $mo(i) \geq var$
$mo'(k) = mo(k)$ **otherwise**

2. $p - n < k \leq p$

if $mo(i) \geq var$ **then** $mo'(k) = \texttt{LUB}(M_{k-(p-n)}, var)$
else $mo'(k) = M_{k-(p-n)}$.

The implementation is based on a reasoning about the unification process of $t_j = f(t_{(p-n+1)}, \ldots, t_p)$ and t_i, under the assumption that t_i and t_j are unifiable and that $t_{(p-n+1)}, \ldots, t_p$ are distinct variables which do not appear in any other term.
By hypothesis, t_j is of the form $f(t_{(p-n+1)}, \ldots, t_p)$. Therefore, t_i is either a variable y or a compound term $f(u_1, \ldots, u_n)$. Hence two cases must be distinguished in the reasoning. They are however amalgamated in the implementation by means of operation `LUB` on modes, mainly.
Let us consider the case when t_i is a variable (say y). This case is possible only if $var \leq mo(i)$. Then σ is simply $\{y \leftarrow f(t_{(p-n+1)}, \ldots, t_p)\}$. The mode of t_i becomes $\texttt{CONS}(n, var, \ldots, var)$. Variables $t_{(p-n+1)}, \ldots, t_p$ are not instantiated: their mode remains var.
On the contrary, if t_i is the compound term $f(u_1, \ldots, u_n)$, σ is equal to $\{t_{(p-n+1)} \leftarrow u_1, \ldots, t_p \leftarrow u_n\}$ since the $t_{j_1}, \ldots, t_{j_n}$ are new distinct variables. $\texttt{EXTR}(mo(i), n)$ provides the modes $M_1, \ldots, M_n$ of $u_1, \ldots, u_n$ which are the new modes for $t_{j_1}, \ldots, t_{j_n}$. The mode of t_i is not modified.
Now what is the effect of applying σ to any other term t_k? Clearly $t_k\sigma$ differs from t_k only if t_i is a variable y and t_k contains y. (In the other cases, only $t_{j_1}, \ldots, t_{j_n}$ are modified.) This cannot happen if t_k does not share with t_i nor if t_k is a subterm of t_j because t_j should contain y and the unification would fail. In any other cases, if the pattern of t_k is known, we adjust its mode according to the new mode of its subterms; if it is not, t_k becomes $t_k\{y \leftarrow f(t_{(p-n+1)}, \ldots, t_p)\}$. So its new mode is $\texttt{IAT}_2(mo(k), \texttt{CONS}(n, var, \ldots, var))$.

The operation `OMode-COMB` is simply defined as

$$\texttt{OMode} - \texttt{COMB}()(mo, p, \{j_1, \ldots, j_n\}, f) = mo'$$

$$mo'(i) = \begin{cases} mo(i) & \text{if } 1 \leq i < p \\ \texttt{CONS}(n, mo(j_1), \ldots, mo(j_n)) & \text{if } i = p \end{cases}$$

Unification of two variables: Consider the test $\texttt{SH(i,j,k)} = (\texttt{MAYSHARE}(i,k) \vee \texttt{MAYSHARE}(j,k))$. Intuitively, it is satisfied when the term t_k may share either with t_i or with t_j. Let $S = \langle$`MAYSHARE, STRUCTURED, ARITY, ARGUMENT`$\rangle$.

$\texttt{OMode-UNIF}(S)(mo, i, j) = mo'$ is defined as follows ($k \in I_p$):

$$mo'(k) = \begin{cases} mo(k) & \text{if } \neg\texttt{SH}(i,j,k) \\ \texttt{UAT}(mo(i), mo(j)) & \text{if } \texttt{SH}(i,j,k) \;\&\; (k = i \vee k = j); \\ \texttt{RECOM}(mo(k), mo'(k_1), \ldots, mo'(k_n)) & \text{if } \texttt{SH}(i,j,k) \& \; i \neq k \neq j \; \& \\ & \texttt{STRUCTURED}(k) \;\&\; \texttt{ARITY}(k) = n \;\& \\ & \texttt{ARGUMENT}(k,h) = k_h \text{ for } h \in [1..n] \\ \texttt{IAT}(mo(k)) & \text{otherwise.} \end{cases}$$

Let us try to explain the definition above. A subterm having no sharing with subterms i or j is not affected by the unification and keeps its mode. The mode of t_i and t_j is given by the abstract unification of their old modes. If a subterm has sharing with t_i and t_j and has a well-defined pattern, its new mode is given by the recomputing operation on modes given the old mode of the subterm and the new modes of the arguments of its pattern. The recomputing is necessary since subterms t_i and t_j may be reachable from t_k and, since their modes may have been updated, the mode of t_k may need to be updated as well. If a subterm has an undefined pattern, its new mode is given by the abstract instantiation of its old mode. Note that the implementation of the operation

requires a form of bottom-up computation.

Junction of two abstract tuples: Let $mo_1 \in \texttt{OMode}_{p_1} and mo_2 \in \texttt{OMode}_{p_2}$.

$\texttt{OMode-JOIN}()(mo_1, mo_2) = mo$

$$mo(i) = \begin{cases} mo_1(i) & \text{if } 1 \leq i \leq p_1 \\ mo_2(i - p_1) & \text{if } p_1 < i \leq p_1 + p_2. \end{cases}$$

Refinement The following refinement function can be applied to the element $mo \in \texttt{OMode}$.

$\texttt{OMode-REFINE(GROUND)}(mo) = mo'$

$$mo'(i) = \begin{cases} ground & \text{if } \texttt{GROUND}(i) \\ mo(i) & \text{otherwise} \end{cases}$$

5 Experimental Evaluation

In this section, we describe experimental results to indicate the practical interest of our approach. Section 5.1 contains some preliminaries. Section 5.2 describes the reduction in development effort. Sections 5.3, 5.4, 5.5, and 5.6 discuss respectively the importance of structural information, open operations, refinements, and open structural information. Sections 5.7 describes the overhead of the approach while section 5.8 discusses the feasibility of designing complex abstract domains. All computation times are given on a Sun SS 10/30.

5.1 Preliminaries

In this section, we quickly review the program tested and the algorithm used.

5.1.1 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 [37] and implements an alpha-beta search procedure. The program `press` is an equation-solver program taken from [37] 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 [38]. 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 [17] 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).

5.1.2 The Fixpoint Algorithm

As mentioned previously, our contributions are independent from the fixpoint algorithm used for the analysis. The experimental results described here make use of the algorithm GAIA [24] which associates with each predicate p in the program a set of tuples $(\beta_{in}, p, \beta_{out})$ where β_{in} and β_{out} represent respectively the input and abstract substitutions. This granularity is appropriate for multiple specializations [41]. A finer granularity would be obtained by using the algorithms presented in [14, 39, 18].

5.1.3 The Implementation

The above theoretical ideas have been integrated in a system organized around a number of abstract data types (ADT):

- an ADT for the pattern domain;
- an ADT for the $\Re$-domain;
- an ADT for each of the open domains.

The pattern domain ADT is the sole interface to the fixpoint algorithm. Its implementation uses the operations available in the $\Re$-domain ADT. The $\Re$-domain ADT is the sole interface to the pattern domain ADT. It provides an implementation of the abstract operations whose specifications have been given previously. It also contains queries which are obtained by combining the queries of the subdomains. The open domain ADTs contain both open operations and queries. The open operations use the queries supported by the $\Re$-domain. As a consequence, these ADT have no knowledge of each other and can only benefit from the other domains through the $\Re$-domain ADT.

5.2 On the Development Effort

In this section, we give some ideas about the effort necessary to produce a sophisticated domain, i.e. 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 +

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

Table 6: Accuracy of the Analysis on Inputs: Comparison of Prop and Pat(Prop)

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

Table 7: Accuracy of the Analysis on Outputs: Comparison of Prop and Pat(Prop)

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$\otimes$OPS) needs only to produce about 10% of the overall code. As should be clear, our approach reduces the development effort substantially. It should also be clear 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.

5.3 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. [20, 26]). We will not repeat those results here. Rather we show that even for very accurate domains such as Prop, preserving structural information improves accuracy on practical programs.

Tables 6 and 7 compare Prop and Pat(Prop) for the input and output patterns and only report the programs for which there are some differences. The results indicate that Pat(Prop) improves on Prop 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(Prop). Note also that the increase in precision is substantial for the press program.

Table 8 depicts the efficiency results of Pat(Prop) and compares them to Prop. The times of the analyses with Pat(Prop) are reasonable, yet they are substantially slower (about 22 times) than Prop. This indicates clearly a tradeoff between efficiency and accuracy in this case.

For completeness, we also consider the use of Pat(Prop) for online analysis (see [16] 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. Prop and Pat(Prop) 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. For instance, in Prop, append(x_1,x_2,x_3) returns $x_3 \Leftrightarrow x_2 \wedge x_1$, and 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.

As far as the accuracy is concerned, the quality of the results is rather surprising. On all

	Prop			Pat(Prop)			Pat(Prop)/Prop		
	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 8: The Domain Pat(Prop): Efficiency Results

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 9: On-line Analysis: Efficiency Results of Pat(Prop)

programs, the specialization of the on-line analysis with the input pattern gives the same result for the top-level goal than the traditional analysis with `Pat(Prop)`. This indicates that `Pat(Prop)` is really a domain of choice for on-line analysis.

Table 9 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.

5.4 On the Importance of Open Operations

The purpose of this section is to investigate the importance of open operations and to find out whether refinement operations can recover the loss of information coming from a direct product. We use the domain `OPAT(OMode⊗OPS)` for the experimental results in its standard version (denoted by `S`) in the tables and in a modified version (denoted `NQ`) where `OMode` and `OPS` can only interact through the refinement operations. Note that queries to structural information are allowed in both the operations and the refinements. Removing the queries to structural information is studied in a later section.

The accuracy results are depicted in Tables 10 and 11. In the tables, `N` is the number of arguments of all predicates, `loss` is the number of arguments for which `NQ` loses precision compared to `S`, and `%loss` is the same measure in percentage. The number of free and ground arguments are also reported for both versions as well as their ratios. Finally, the number of pairs of arguments which possibly share are also given for both programs as well as their ratios. All these results concerns only the top-level arguments and not nested terms and the modes compared are those described previously in this paper.

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`. 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 in Table 12 show that `S` is slightly more efficient than `NQ` in the average, demonstrating that open operations are particularly appropriate. Note that `S` is about twice faster than `NQ` on `read`.

5.5 On the Importance of Refinements

The purpose of this section is to investigate the importance of refinements in conjunction with open operations and 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` in the tables) and in a modified version (denoted `NR`) where no refinement operations are used. The accuracy results are depicted in Table 13 only for sharing since the other components have the same accuracy. The notations are consistent with those used in the previous tables. The results for the inputs are the same in both versions.

The accuracy results show that refinement operations can improve substantially the sharing component on the output patterns. The remaining results remain unchanged. Note that more improvement may appear if information on all subterms is computed. Nevertheless refinement

		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 10: The Importance of Open Operations: Input Results

		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 11: The Importance of Open Operations: Output Results

	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 12: The Importance of Open Operations: Efficiency Results

		sharing		
	N	S	NR	%NR/S
cs	94	38	91	239.47
disj	60	22	66	300.00
gabriel	59	44	44	104.55
kalah	123	44	111	252.27
peep	63	24	80	333.33
pg	31	11	31	281.82
plan	32	13	35	269.23
press1	143	169	186	110.06
press2	143	44	185	420.45
qsort	9	5	8	160.00
queens	11	4	9	225.00
read	122	98	129	131.63
Mean				229.06

Table 13: The Importance of Refinements: Output Results

	S (Standard Version)			NR (No Refinement)			Ratios (NR/S)		
	Time	GI	CI	Time	GI	CI	Time	GI	CI
cs	4.75	85	168	6.21	98	198	1.31	1.15	1.18
disj	2.45	68	134	3.27	77	149	1.33	1.13	1.11
gabriel	1.64	81	190	1.77	84	196	1.08	1.04	1.03
kalah	4.41	117	236	5.51	146	295	1.25	1.25	1.25
peep	5.03	94	530	4.56	102	463	0.91	1.09	0.87
pg	0.66	38	80	1.21	51	108	1,83	1.34	1.35
plan	0.57	36	78	0.91	55	114	1.60	1.53	1.46
press1	24.06	554	1841	42.66	612	2069	1.77	1.10	1.12
press2	6.90	212	707	10.98	287	998	1.59	1.35	1.41
qsort	0.16	13	28	0.19	13	28	1.19	1.00	1.00
queens	0.15	15	29	0.13	17	33	0.87	1.13	1.14
read	10.95	209	938	12.10	237	1284	1.11	1.13	1.37
Mean							1.29	1.22	1.23

Table 14: The Importance of Refinement Operations: Efficiency Results

operations 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 in Table 14 indicate the benefit of refinement operations as well. GI and CI denote the number of goal and clause iterations respectively. 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.

5.6 On the Importance of Open Structural Information

The purpose of this section is to investigate the importance of allowing the subdomain operations to query structural information. We compare the domains OPAT(OMode⊗OPS) and PAT(OMode⊗OPS) and refer to them by S and NS in the tables. PAT(OMode⊗OPS) does not make use of queries to structural information. The accuracy results are depicted in Tables 15 and 16 and the notation is consistent with the previous tables.

The accuracy results demonstrate the importance of opening the generic pattern domain. As far as input patterns are concerned, NS loses in the average about 32% accuracy for modes, 8% for freeness, 51% for groundness, and produces 2.22 more sharing than S. As far as output patterns are concerned, NS loses in the average about 49% accuracy for modes, 53% for freeness, 51% for groundness, and produces substantially more sharing than S. This clearly demonstrates the benefits of opening the generic domain to the subdomains.

The efficiency results in Table 17 show that S is about 1.5 more efficient than NS in the average, demonstrating that structural information not only improves precision but also leads to a better efficiency.

		modes		freeness			groundness			sharing		
	N	loss	%loss	S	NS	%(S-NS)/S	S	NS	%(S-NS)/S	S	NS	%NS/S
cs	94	34	36.17	37	37	0.00	56	22	60.71	38	88	231.58
disj	60	28	46.67	21	21	0.00	38	11	71.05	22	64	290.91
gabriel	59	0	0.00	18	18	0.00	18	18	0.00	44	44	100.00
kalah	123	50	40.65	39	39	0.00	79	34	56.96	44	96	218.18
peep	63	18	28.57	15	15	0.00	39	21	46.15	24	69	287.50
pg	31	14	45.16	6	6	0.00	20	8	60.00	11	28	254.55
plan	32	16	50.00	6	6	0.00	20	5	75.00	13	35	269.23
press1	143	2	1.40	36	36	0.00	15	13	13.33	169	171	101.18
press2	143	88	61.54	36	36	0.00	99	13	86.87	44	171	388.64
qsort	9	3	33.33	3	0	100.00	4	1	75.00	5	9	180.00
queens	11	5	45.45	4	4	0.00	7	2	71.43	4	10	250.00
read	122	1	0.82	46	46	0.00	34	34	0.00	98	98	100.00
Mean			32.48			8.33			51.38			222.65

Table 15: The Importance of Open Structural Information: Input Results

		modes		groundness			sharing		
	N	loss	%loss	S	NS	%(S-NS)/S	S	NS	%NS/S
cs	94	62	65.96	94	32	65.96	0	110	∞
disj	60	31	51.67	60	29	51.67	0	48	∞
gabriel	59	0	0.00	22	22	0.00	55	55	100.00
kalah	123	69	56.10	121	54	55.37	2	99	4950.00
peep	63	26	41.27	53	27	49.06	11	75	681.82
pg	31	22	70.97	31	9	70.97	0	31	∞
plan	32	24	75.00	31	8	74.19	1	40	4000.00
press1	143	8	5.59	40	32	20.00	179	190	106.15
press2	143	109	76.22	140	32	77.14	3	190	6333.33
qsort	9	5	55.56	6	1	83.33	4	11	275
queens	11	9	81.82	11	2	81.82	0	13	∞
read	122	5	4.10	74	69	6.76	86	91	105.81
Mean			48.69			53.02			∞

Table 16: The Importance of Open Structural Information: Output Results

	S (Standard Version)			NR (No Open Pattern)			Ratios (NR/S)		
	Time	GI	CI	Time	GI	CI	Time	GI	CI
cs	4.75	85	168	6.29	107	215	1.32	1.26	1.28
disj	2.45	68	134	2.72	74	144	1.11	1.09	1.07
gabriel	1.64	81	190	1.79	81	189	1.09	1.00	0.99
kalah	4.41	117	236	6.17	150	309	1.40	1.28	1.31
peep	5.03	94	530	4.76	98	472	0.95	1.04	0.89
pg	0.66	38	80	0.97	47	100	1.47	1.24	1.25
plan	0.57	36	78	1.37	64	134	2.40	1.78	1.72
press1	24.06	554	1841	19.88	460	1626	0.83	0.83	0.88
press2	6.90	212	707	20.72	472	1698	3.00	2.23	2.40
qsort	0.16	13	28	0.27	15	34	1.69	1.15	1.21
queens	0.15	15	29	0.26	19	37	1.73	1.27	1.28
read	10.95	209	938	11.04	200	861	1.01	0.96	0.92
Mean							1.50	1.26	1.27

Table 17: The Importance of Open Structural Operations: Efficiency Results

5.7 On the Overhead of The Approach

In this section, we study the overhead of our approach compared to a direct implementation of our domain [24]. Our approach introduces mainly three forms of overheads:

1. global operations: the generic pattern domain has provisions to accommodate global information on variables which complicates the operations when only local information is used;
2. memory management: the approach allocates and deallocates memory with a much smaller granularity;
3. queries: the query mechanism introduces an additional layer necessary to combine the domain.

The experimental results were obtained on the domain `OPAT(OMode⊗OPS)`. This domain is very appropriate for measuring the overhead because it tests the three types of overhead. It particular, it does not require global information (like in `Prop`) and can be built with simplified versions of all abstract operations removing many needs to projections, unifications, and renamings on the $\Re$-domain. The times given in this section should be interpreted with care, since the implementation has not be tuned with the same care as the direct implementation. In particular, the memory management has not been changed to accommodate the new needs of the generic domain. Table 18 describes the experimental results. They 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. Moreover, we believe that the overhead can be significantly reduced by improving memory management, caching all queries whenever appropriate, and specializing the implementation when the full generality is not needed. This is obviously an important topic for further research.

	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 18: The Overhead of The Approach: Efficiency Results

		modes		freeness			groundness			sharing		
	N	loss	%loss	C	S	%(C-S)/C	C	S	%(C-S)/S	C	S	%S/C
press1	143	86	60.14	36	36	0.00	99	15	84.15	44	169	384.09

Table 19: On the Feasibility of Complex Domains: Input Results

5.8 On the Feasibility of Complex Domains

In this section, we show how the approach can be used for OPAT(OProp⊗OMode⊗OPS) a very complex domain combining structural information, modes, sharing, and Prop. This domain is probably one of the most complex domains built for the abstract interpretation of Prolog and would require substantial effort for a direct implementation. It obviously combines both local and global domains.

The accuracy results are depicted in Tables 19 and 20. In the following we use C to denote OPAT(OProp⊗OMode⊗OPS) and S to denote OPAT(OMode⊗OPS). The improvements occur only in programs using difference lists or structures, i.e. press, peep, and qsort, since the accuracy of OPAT(OMode⊗OPS) is optimal or close to optimal on all other programs [24]. The efficiency results indicate that C is about 5.6 slower than S. This time can certainly be reduced by caching the queries to OProp, since the groundness queries in OProp are rather time-consuming. Note that C does almost always less iterations than S

		modes		freeness			groundness			sharing		
	N	loss	%loss	C	S	%(C-S)/C	C	S	%(C-S)/S	C	S	%S/C
peep	63	1	1.59	0	0	0.00	54	53	1.85	9	11	122.22
press1	143	101	70.63	0	0	0.00	140	40	71.43	3	179	5966.67
qsort	9	1	11.11	0	0	0.00	7	6	14.29	3	4	133.33

Table 20: On the Feasibility of Complex Domains: Output Results

	OPAT(OMode⊗OPS)			OPAT(OProp⊗OMode⊗OPS)			Ratios		
	Time	GI	CI	Time	GI	CI	Time	GI	CI
cs	4.75	85	168	14.26	85	168	3.00	1.00	1.00
disj	2.45	68	134	7.67	68	134	3.13	1.00	1.00
gabriel	1.64	81	190	11.02	82	193	6.72	1.01	1.02
kalah	4.41	117	236	17.75	117	236	4.02	1.00	1.00
peep	5.03	94	530	25.40	96	538	5.05	1.02	1.02
pg	0.66	38	80	2.38	38	80	3.61	1.00	1.00
plan	0.57	36	78	2.53	36	78	4.44	1.00	1.00
press1	24.06	554	1841	28.51	207	674	1.18	0.37	0.37
press2	6.90	212	707	29.78	213	710	4.32	1.00	1.00
qsort	0.16	13	28	0.74	13	28	4.63	1.00	1.00
queens	0.15	15	29	0.39	15	29	2.60	1.00	1.00
read	10.95	209	938	263.87	239	1073	24.10	1.14	1.14
Mean							5.57	0.96	0.96

Table 21: The Feasibility of Complex Domains: Efficiency Results

6 Conclusion

With the emergence of efficient generic fixpoint algorithms, most of the difficulty in designing an analysis lies in the abstract domain. This paper studied how the design process can be supported both at the conceptual and implementation level. It introduced two novel ideas: a generic domain for structural information and a notion of open product.

The generic pattern domain automatically upgrades a domain with structural information abstracting away structural information while improving accuracy. The generic domain assumes a small number of operations on the domain and the support consists in a number of generic algorithms which achieves the abstract operations needed by a fixpoint algorithm. The generic pattern domain is independent from the fixpoint algorithm and can be used with algorithms working at different granularities.

The open product is a new way of combining abstract domains which can be used to implement or approximate the reduced product of the Cousots. It is based on the notions of queries and open operations, i.e. operations which are parametrized by tests providing information not necessarily maintained in the domain. The consistency of the open product has been charactized formally and we have shown how the idea of refinements, developed independently by [6], is naturally formalized in this framework.

The two contributions have been implemented in a system modelled along the theory and organized in a number of abstract data types. We have shown how very complex domains can be built in the system with minimal development effort (10-20% of the overall code). The significance of all the contributions (e.g. structural information, open information, refinements, and open structural information) have all been characterized experimentally by integrating the resulting domains inside the GAIA system.

Future research will be devoted to enhancements of the system to provide generic subdomains and other domains such Janssens' type system. Graphical support to design abstract domains should further simplify the development time and the automation of the integration. Finally,

implementation issues should be studied with care to improve further the efficiency.

Acknowledgements

This research was 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 CNR, Prog. Fin. "Sistemi Informatici e Calcolo Parallelo" under grant n.91.00026.69.

References

[1] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. (To appear in *ACM Transactions on Programming Languages and Systems*).

[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. 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

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

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

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

[32] 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.

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

[34] 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.

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

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

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

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

[39] 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.

[40] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The impact of Granularity in Abstract Interpretation of Prolog. Technical report, CS Department, Brown University, 1993. Forthcoming.

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

Contents

1 Introduction **1**

2 Preliminaries **3**
- 2.1 Substitutions . . . 3
- 2.2 Operations . . . 4
- 2.3 Abstractions . . . 5

3 The Generic Pattern Domain `Pat(`$\Re$`)` **5**
- 3.1 Overview . . . 5
- 3.2 Semantics . . . 6
  - 3.2.1 Pattern Component . . . 7
  - 3.2.2 Same Value Component . . . 7
  - 3.2.3 The $\Re$-component . . . 7
  - 3.2.4 The Concretization Function . . . 9
- 3.3 Implementation: Abstract Operations . . . 10
  - 3.3.1 The Natural Transformation . . . 11
  - 3.3.2 Operation `EXTC`$(\beta, c) = \beta'$ . . . 12
  - 3.3.3 Operation `RESTRC`$(\beta, c) = \beta'$ . . . 12
  - 3.3.4 Unification . . . 13
  - 3.3.5 Operation `SPECAT`$(i, j, \delta) = \delta'$ . . . 16
  - 3.3.6 Operation `UNIF`$(i, j, \delta) = \delta'$ . . . 16
  - 3.3.7 Operation `AI_VAR`$(\beta) = \beta'$ . . . 17
  - 3.3.8 Operation `AI_FUNC`$(f, \beta) = \beta'$ . . . 18
  - 3.3.9 Operation `EXTG`$(g, \beta_1, \beta_2) = \beta'$ . . . 18
  - 3.3.10 Operation `UNION`$(\beta_1, \beta_2) = \beta$ . . . 19
- 3.4 Application: The Domain `Pat(Prop)` . . . 20
  - 3.4.1 Domain . . . 20
  - 3.4.2 Operations . . . 21

4 Open Product **21**
- 4.1 Overview . . . 22
- 4.2 Semantics . . . 23
  - 4.2.1 Open Interpretations . . . 23
  - 4.2.2 Open Abstract Interpretations . . . 24
  - 4.2.3 Open Product . . . 25
  - 4.2.4 Refined Open Product . . . 27
- 4.3 Application I: $\Re$ as an Open Product . . . 28
  - 4.3.1 Overview . . . 28
  - 4.3.2 The Open Interpretation `OProp` . . . 29
  - 4.3.3 The Open Interpretation `OPS` . . . 29
  - 4.3.4 The Open Interpretation `OLIN` . . . 31
  - 4.3.5 The open product $\Re =$ `OProp` $\otimes$ `OLIN` $\otimes$ `OPS`. . . . 33
- 4.4 Application II: The Open Generic Pattern Domain `OPat`$\{\Re\}$ . . . 34
  - 4.4.1 Overview . . . 34

4.4.2 Queries . . . 34
4.4.3 An Auxiliary Domain of Modes . . . 35
4.4.4 The Open Pattern Domain `OPat{OMode}` . . . 38

5 Experimental Evaluation **42**
5.1 Preliminaries . . . 42
5.1.1 The Programs Tested . . . 42
5.1.2 The Fixpoint Algorithm . . . 43
5.1.3 The Implementation . . . 43
5.2 On the Development Effort . . . 43
5.3 On the Importance of Structural Information . . . 44
5.4 On the Importance of Open Operations . . . 46
5.5 On the Importance of Refinements . . . 46
5.6 On the Importance of Open Structural Information . . . 49
5.7 On the Overhead of The Approach . . . 51
5.8 On the Feasibility of Complex Domains . . . 52

6 Conclusion **53**