Efficient Structural Information Analysis for Real CLP Languages*

Roberto Bagnara[1], Patricia M. Hill[2], and Enea Zaffanella[1]

[1] Department of Mathematics, University of Parma, Italy.
{bagnara,zaffanella}@cs.unipr.it
[2] School of Computing, University of Leeds, U. K.
hill@comp.leeds.ac.uk

Abstract. We present the rational construction of a generic domain for structural information analysis of CLP languages called Pattern($\mathcal{D}^\sharp$), where the parameter $\mathcal{D}^\sharp$ is an abstract domain satisfying certain properties. Our domain builds on the parameterized domain for the analysis of logic programs `Pat(`$\Re$`)`, which is due to Cortesi et al. However, the formalization of our CLP abstract domain is independent from specific implementation techniques: `Pat(`$\Re$`)` (suitably extended in order to deal with CLP systems omitting the occur-check) is one of the possible implementations. Reasoning at a higher level of abstraction we are able to appeal to familiar notions of unification theory. This higher level of abstraction also gives considerable more latitude for the implementer. Indeed, as demonstrated by the results summarized here, an analyzer that incorporates structural information analysis based on our approach can be highly competitive both from the precision *and*, contrary to popular belief, from the efficiency point of view.

1 Introduction

Most interesting CLP languages [16] offer a constraint domain that is an amalgamation of a domain of syntactic trees — like the classical domain of finite trees (also called the *Herbrand* domain) or the domain of rational trees [9] — with a set of "non-syntactic" domains, like finite domains, the domain of rational numbers and so forth. The inclusion of uninterpreted functors is essential for preserving Prolog programming techniques. Moreover, the availability of syntactic constraints greatly contributes to the expressive power of the overall language. When syntactic structures can be used to build aggregates of interpreted terms one can express, for instance, "records" or "unbounded containers" of numerical quantities.

From the experience gained with the first prototype version of the China data-flow analyzer [1] it was clear that, in order to attain a significant precision

* This work has been partly supported by MURST project "Certificazione automatica di programmi mediante interpretazione astratta." Some of this work was done during a visit of the first and third authors to Leeds, funded by EPSRC under grant M05645.

in the analysis of numerical constraints in CLP languages, one must keep at least part of the uninterpreted terms in concrete form. Note that almost any analysis is more precise when this kind of structural information is retained to some extent: in the case mentioned here the precision loss was just particularly acute. Of course, structural information is very valuable in itself. When exploited for optimized compilation it allows for enhanced clause indexing and simplified unification. Moreover, several program verification techniques are highly dependent on this kind of information.

Cortesi et al. [10, 11], after the work of Musumbu [21], put forward a very nice proposal for dealing with structural information in the analysis of logic programs. Using their terminology, they defined a generic abstract domain $\mathtt{Pat}(\Re)$ that automatically upgrades a domain $\Re$ (which must support a certain set of elementary operations) with structural information.

As far as the overall approach is concerned, we extend the work described in [11] by allowing for the analysis of any CLP language [16]. Most importantly, we do *not* assume that the analyzed language performs the *occur-check* in the unification procedure. This is an important contribution, since the vast majority of *real* (i.e., implemented) CLP languages (in particular, almost all Prolog systems) do omit the occur-check, either as a mere efficiency measure or because they are based upon a theory of extended rational trees [9]. We describe a generic construction for structural analysis of CLP languages. Given an abstract domain $\mathcal{D}^\sharp$ satisfying a small set of very reasonable and weak properties, the structural abstract domain $\mathrm{Pattern}(\mathcal{D}^\sharp)$ is obtained automatically by means of this construction. In contrast to [11], where the authors define a specific implementation of the generic structural domain (e.g., of the representation of term-tuples), the formalization of $\mathrm{Pattern}(\cdot)$ is implementation-independent: $\mathtt{Pat}(\Re)$ (suitably extended in order to deal with CLP languages and with the occur-check problem) is a possible base for the implementation. Reasoning at a higher level of abstraction we are able to appeal to familiar notions of unification theory [18]. One advantage is that we can identify an important parameter (a common anti-instance function) that gives some control over the precision and computational cost of the resulting structural domain. In addition, we believe our implementation-independent treatment can be more easily adapted to different analysis frameworks/systems.

One of the merits of $\mathtt{Pat}(\Re)$ is to define a generic implementation that works on any domain $\Re$ that provides a certain set of elementary, fine-grained operations. Because of the simplicity of these operations it is particularly easy to extend an existing domain in order to accommodate them. However, this simplicity has a high cost in terms of efficiency: the execution of many isolated small operations over the underlying domain is much more expensive than performing few macro-operations where global effects can be taken into account. The operations that the underlying domain must provide are thus more complicated in our approach. However, this extra complication and the higher level of abstraction give considerable more latitude for the implementer. Indeed, as demonstrated by the results summarized here, an analyzer that incorporates structural information analysis based on our approach can be highly competitive both from the precision *and* the efficiency point of view. One of the contributions of this paper is that it disproves the common belief (now reinforced by [8]) whereby abstract domains enhanced with structural information are inherently inefficient.

The paper is structured as follows: Section 2 introduces some basic concepts and the notation that will be used in the paper; Section 3 presents the main ideas behind the tracking of explicit structural information for the analysis of CLP languages; Section 4 introduces the $\mathcal{D}^\sharp$ and Pattern($\mathcal{D}^\sharp$) domains and explains how an abstract semantics based on $\mathcal{D}^\sharp$ can systematically be upgraded to one on Pattern($\mathcal{D}^\sharp$); Section 5 summarizes the extensive experimental evaluation that has been conducted to validate the ideas presented in this paper; Section 6 presents a brief discussion of related work and, finally, Section 7 concludes with some final remarks.

2 Preliminaries

Let U be a set. The cardinality of U is denoted by $|U|$. We will denote by U^n the set of n-tuples of elements drawn from U, whereas U^* denotes $\bigcup_{n\in\mathbb{N}} U^n$. Elements of U^* will be referred to as *tuples* or as *sequences*. The *empty sequence*, i.e., the only element of U^0, is denoted by ε. Throughout the paper all variables denoting sequences will be written with a "bar accent" like in $\bar{s}$. For $\bar{s} \in U^*$, the *length* of $\bar{s}$ will be denoted by $|\bar{s}|$. The concatenation of the sequences $\bar{s}_1, \bar{s}_2 \in U^*$ is denoted by $\bar{s}_1 :: \bar{s}_2$. For each $\bar{s} \in U^\star$ and each set $X \in \wp_{\mathrm{f}}(U)$, the sequence $\bar{s} \setminus X$ is obtained by removing from $\bar{s}$ all the elements that appear in X. The *projection mappings* $\pi_i \colon U^n \to U$ are defined, for $i = 1, \ldots, n$, by $\pi_i\bigl((e_1, \ldots, e_n)\bigr) = e_i$. We will also use the liftings $\pi_i \colon \wp(U^n) \to \wp(U)$ given by $\pi_i(S) = \bigl\{\, \pi_i(\bar{s}) \bigm| \bar{s} \in S \,\bigr\}$. If a sequence $\bar{s}$ is such that $|\bar{s}| \geq i$, we let $\mathrm{prefix}_i(\bar{s})$ denote the sequence of the first i elements of $\bar{s}$.

Let *Vars* denote a denumerable and totally ordered set of variable symbols. We assume that *Vars* contains (among others) two infinite, disjoint subsets: $\mathbf{z}$ and $\mathbf{z}'$. Since *Vars* is totally ordered, $\mathbf{z}$ and $\mathbf{z}'$ are as well. Thus we assume $\mathbf{z} = (Z_1, Z_2, Z_3, \ldots$ and $\mathbf{z}' = (Z_1', Z_2', Z_3', \ldots$. If $W \subseteq \mathit{Vars}$ we will denote by $\mathcal{T}_W$ the set of terms with variables in W. For any term or a tuple of terms t we will denote the set of variables occurring in t by $\mathit{vars}(t)$. We will also denote by $\mathit{vseq}(t)$ the sequence of first occurrences of variables that are found on a depth-first, left-to-right traversal of t. For instance, $\mathit{vseq}\bigl(\bigl(f(g(X), Y), h(X)\bigr)\bigr) = (X, Y)$.

We implement the "renaming apart" mechanism by making use of two strong normal forms for tuples of terms. Specifically, the set of *n-tuples in $\mathbf{z}$-form* is given by $\mathbf{T}^n_{\mathbf{z}} = \bigl\{\, \bar{t} \in \mathcal{T}^n_{\mathit{Vars}} \bigm| \mathit{vseq}(\bar{t}) = \bigl(Z_1, Z_2, \ldots, Z_{|\mathit{vars}(\bar{t})|}\bigr) \,\bigr\}$. The set of all the tuples in $\mathbf{z}$-form is denoted by $\mathbf{T}^*_{\mathbf{z}}$. The definitions for $\mathbf{T}^n_{\mathbf{z}'}$ and $\mathbf{T}^*_{\mathbf{z}'}$ are obtained in a similar way, by replacing $\mathbf{z}$ with $\mathbf{z}'$. There is a useful device for toggling between $\mathbf{z}$- and $\mathbf{z}'$-forms. Let $\bar{t} \in \mathbf{T}^n_{\mathbf{z}} \cup \mathbf{T}^n_{\mathbf{z}'}$ and $\bigl|\mathit{vars}(\bar{t})\bigr| = m$. Then $\bar{t}' = \bar{t}[Z_1'/Z_1, \ldots, Z_m'/Z_m]$, if $\bar{t} \in \mathbf{T}^n_{\mathbf{z}}$, and $\bar{t}[Z_1/Z_1', \ldots, Z_m/Z_m']$, if $\bar{t} \in \mathbf{T}^n_{\mathbf{z}'}$. Notice that $\bar{t}'' = \bigl(\bar{t}'\bigr)' = \bar{t}$.

When $\bar{V} \in \mathit{Vars}^m$ and $\bar{t} \in \mathcal{T}^m_{\mathit{Vars}}$ we use $[\bar{t}/\bar{V}]$ as a shorthand for the substitution $\bigl[\pi_1(\bar{t})/\pi_1(\bar{V}), \ldots, \pi_m(\bar{t})/\pi_m(\bar{V})\bigr]$, if $m > 0$, and to denote the empty substitution if $m = 0$. If $\mathit{vars}(\bar{t}) \cap \bar{V} = \varnothing$, then $[\bar{t}/\bar{V}]$ is *idempotent*. Suppose that $\bar{s} = (s_1, \ldots, s_m) \in \mathcal{T}^m_{\mathit{Vars}}$ and $\bar{t} = (t_1, \ldots, t_m) \in \mathcal{T}^m_{\mathit{Vars}}$, then, $\bar{s} = \bar{t}$ denotes $(s_1 = t_1, \ldots, s_m = t_m)$. It is also useful to sometimes regard a substitution $[\bar{t}/\bar{V}]$ as the finite set of equations $\bar{V} = \bar{t}$. A couple of observations are useful for what follows. If $\bar{s} \in \mathbf{T}^*_{\mathbf{z}}$ and $\bar{u} \in \mathbf{T}^{|\mathit{vars}(\bar{s})|}_{\mathbf{z}}$, then $\bar{s}'\bigl[\bar{u}/\mathit{vseq}(\bar{s}')\bigr] \in \mathbf{T}^*_{\mathbf{z}}$. Moreover $\mathit{vseq}\bigl(\bar{s}'\bigl[\bar{u}/\mathit{vseq}(\bar{s}')\bigr]\bigr) = \mathit{vseq}(\bar{u})$.

The logical theory underlying a CLP constraint system [16] is denoted by $\mathfrak{T}$. To simplify the notation, we drop the outermost universal quantifiers from (closed) formulas so that if F is a formula with free variables $\bar{Z}$, then we write $\mathfrak{T} \models F$ to denote the expression $\mathfrak{T} \models \forall \bar{Z} : F$.

The notation $f \colon A \rightarrowtail B$ signifies that f is a *partial* function from A to B.

3 Making the Herbrand Information Explicit

A quite general picture for the analysis of a CLP language is as follows. We want to describe a (possibly infinite) set of constraint stores over a tuple of *variables of interest* $\bar{V} = (V_1, \ldots, V_k)$. Each constraint store can be represented, at some level of abstraction, by a formula of the kind $\exists_\Delta \,.\, \bigl((\bar{V} = \bar{t}) \wedge C\bigr)$, where $(\bar{V} = \bar{t})$, with $\bar{t} \in \mathcal{T}^k_{\mathit{Vars}}$, is a system of Herbrand equations in solved form, $C \in \mathcal{C}^\flat$ is a constraint on the concrete constraint domain $\mathcal{C}^\flat$, and the set $\Delta = \mathit{vars}(C) \cup \mathit{vars}(\bar{t})$ is such that $\Delta \cap \bar{V} = \varnothing$. Roughly speaking, C limits the values that the quantified variables occurring in $\bar{t}$ can take. Notice that this treatment does not exclude the possibility of dealing with domains of rational trees: the non-Herbrand constraints will simply live in the constraint component. For example, the constraint store resulting from execution of the SICStus goal '`?- X = f(a, X)`' may be captured by $\exists X \,.\, \bigl(\{V_1 = X\} \wedge X = f(a, X)\bigr)$ but also by $\exists X \,.\, \bigl(\bigl\{V_1 = f(a, X)\bigr\} \wedge X = f(a, X)\bigr)$.

Once variables $\bar{V}$ have been fixed, the Herbrand part of the constraint store can be represented as a k-tuple of terms. We are thus assuming a concrete domain where the Herbrand information is explicit and other kinds of information are captured by some given constraint domain $\mathcal{C}^\flat$. For instance, if the target language of the analysis is CLP($\mathcal{R}$) [17], $\mathcal{C}^\flat$ may encode conjunctions of equations and inequations over arithmetic expressions, the mechanisms for delaying non-linear constraints, and other peculiarities of the arithmetic part of the language. We assume constraints are modeled by logical formulas, so that it makes sense to talk about the *free variables* of $C^\flat \in \mathcal{C}^\flat$, denoted by $FV(C^\flat)$. These are the variables that the constraint solver makes visible to the Herbrand engine, all the other variables being restricted in scope to the solver itself. Since we want to characterize any set of constraint stores, our concrete domain is

$$\mathcal{D}^\flat = \bigcup_{n \in \mathbb{N}} \wp\Bigl(\bigl\{\, (\bar{s}, C^\flat) \bigm| \bar{s} \in \mathbf{T}^n_{\mathbf{z}}, C^\flat \in \mathcal{C}^\flat, FV(C^\flat) \subseteq \mathit{vars}(\bar{s}) \,\bigr\}\Bigr)$$

partially ordered by subset inclusion.

$$
\begin{tikzcd}
\wp\bigl(\mathbf{T}^*_{\mathbf{z}} \times \mathcal{C}^\flat\bigr) \arrow[r, "\alpha"] \arrow[d, shift left, "\Phi_\phi^{-1}"] & \mathcal{D}^\sharp \\
\mathbf{T}^*_{\mathbf{z}} \times \wp\bigl(\mathbf{T}^*_{\mathbf{z}} \times \mathcal{C}^\flat\bigr) \arrow[r, "(\mathrm{id},\alpha)"'] \arrow[u, shift left, "\Phi_\phi"] & \mathbf{T}^*_{\mathbf{z}} \times \mathcal{D}^\sharp \arrow[u, dotted, "\alpha'"']
\end{tikzcd}
$$

Fig. 1. Upgrading a domain with structural information.

An abstract interpretation [12] of $\mathcal{D}^\flat$ can be specified by choosing an abstract domain $\mathcal{D}^\sharp$ and a suitable abstraction function $\alpha\colon \mathcal{D}^\flat \to \mathcal{D}^\sharp$. If $\mathcal{D}^\sharp$ is not able to encode enough structural information from $\mathcal{C}^\flat$ so as to achieve the desired precision, it is possible to improve the situation by keeping some Herbrand information explicit. One way of doing that is to perform a change of representation for $\mathcal{D}^\flat$ and use the new representation as the basis for abstraction. The new representation is obtained by factoring out some common Herbrand information. The meaning of 'some' is encoded by a function.

Definition 1. (Common anti-instance function.) *For each $n \in \mathbb{N}$, a function $\phi\colon \wp(\mathbf{T}^n_{\mathbf{z}}) \to \mathbf{T}^n_{\mathbf{z}'}$ is called a* common anti-instance function *if and only if the following holds: whenever $T \in \wp(\mathbf{T}^n_{\mathbf{z}})$, if $\phi(T) = \bar{r}'$ and $\bigl|vars(\bar{r})\bigr| = m$ with $m \geq 0$, then $\forall \bar{t} \in T : \exists \bar{u} \in \mathbf{T}^m_{\mathbf{z}} \,.\, \bar{r}'\bigl[\bar{u}/vseq(\bar{r}')\bigr] = \bar{t}$. In words, $\phi(T)$ is an* anti-instance [18]*, in $\mathbf{z}'$-form, of each $\bar{t} \in T$.*

Any choice of ϕ induces a function $\Phi_\phi\colon \mathcal{D}^\flat \to \mathbf{T}^*_{\mathbf{z}} \times \mathcal{D}^\flat$, which is given, for each $E^\flat \in \mathcal{D}^\flat$, by $\Phi_\phi(E^\flat) = \bigl(\bar{s}, \bigl\{\, (\bar{u}, G^\flat) \bigm| (\bar{t}, G^\flat) \in E^\flat, \bar{s}'\bigl[\bar{u}/vseq(\bar{s}')\bigr] = \bar{t} \,\bigr\}\bigr)$, where $\bar{s}' = \phi\bigl(\pi_1(E^\flat)\bigr)$. The corestriction to the image of Φ_ϕ, that is the function $\Phi_\phi\colon \mathcal{D}^\flat \to \Phi_\phi\bigl(\mathcal{D}^\flat\bigr)$, is an isomorphism, the inverse being given, for each $F^\flat \in \mathcal{D}^\flat$, by $\Phi_\phi^{-1}\bigl((\bar{s}, F^\flat)\bigr) = \bigl\{\, \bigl(\bar{s}'\bigl[\bar{u}/vseq(\bar{s}')\bigr], G^\flat\bigr) \bigm| (\bar{u}, G^\flat) \in F^\flat \,\bigr\}$.

So far, we have just chosen a different representation for $\mathcal{D}^\flat$, that is $\Phi_\phi\bigl(\mathcal{D}^\flat\bigr)$. The idea behind structural information analysis is to leave the first component of the new representation (the *pattern component*) untouched, while abstracting the second component by means of α, as illustrated in Figure 1. The dotted arrow indicates a *residual abstraction function* α'. As we will see in Section 4.2, such a function is implicitly required in order to define an important operation over the new abstract domain $\mathbf{T}^*_{\mathbf{z}} \times \mathcal{D}^\sharp$. Notice that, in general, α' does not make the diagram of Figure 1 commute.

This approach has several advantages. First, factoring out common structural information improves the analysis precision, since part of the approximated k-tuples of terms is recorded, *in concrete form*, into the first component of $\mathbf{T}^*_{\mathbf{z}} \times \mathcal{D}^\sharp$. Secondly, the above construction is adjustable by means of the parameter ϕ. The most precise choice consists in taking ϕ to be a *least common anti-instance* (lca) function. For example, the set $E^\flat = \bigl\{\bigl\langle\bigl(s(0), Z_1\bigr), C_1\bigr\rangle, \bigl\langle\bigl(s(s(0)), Z_1\bigr), C_2\bigr\rangle\bigr\}$, is mapped onto $\Phi_{\mathrm{lca}}(E^\flat) = \bigl(\bigl(s(Z_1), Z_2\bigr), \bigl\{\bigl\langle(0, Z_1), C_1\bigr\rangle, \bigl\langle\bigl(s(0), Z_1\bigr), C_2\bigr\rangle\bigr\}\bigr)$, where $C_1, C_2 \in \mathcal{C}^\flat$. At the other end of the spectrum is the possibility of choosing

ϕ so that it returns a k-tuple of distinct variables for each set of k-tuples of terms. This corresponds to a framework where structural information is simply discarded. With this choice, $E^\flat$ would be mapped onto $\bigl((Z_1, Z_2), E^\flat\bigr)$. In-between these two extremes there are a number of possibilities that help to manage the complexity/precision tradeoff. The tuples returned by ϕ can be limited in *depth*, for instance. Another possibility is to limit them in *size*, that is, limiting the number of occurrences of symbols or the number of variables. This flexibility enables the analysis' domains to be designed without considering the structural information: the problem for the domain designers is to approximate the elements of $\wp\bigl(\mathbf{T}_{\mathbf{z}}^k \times \mathcal{C}^\flat\bigr)$ with respect to the property of interest. It does not really matter whether k is fixed by the arity of a predicate or k is the number of variables occurring in a pattern.

4 Parametric Structural Information Analysis

In this section we describe how a complete abstract semantics — which includes an abstract domain plus all the operations needed to approximate the concrete semantics — can be turned into one keeping track of structural information.

We first need some assumptions on the domain $\mathcal{C}^\flat$, which represents the non-Herbrand part of constraint stores. Following [14], it is not at all restrictive to assume that, in order to define the concrete semantics of programs, four operations over $\mathcal{C}^\flat$ need to be characterized. These model the constraint accumulation process, parameter passing, projection, and renaming apart (see also [1, 2] on this subject).

Constraint accumulation is modeled by the binary operator '$\otimes$': $\mathcal{C}^\flat \times \mathcal{C}^\flat \to \mathcal{C}^\flat$ and the unsatisfiability condition in the constraint solver is modeled by the special value $\bot^\flat \in \mathcal{C}^\flat$. Notice that, while '$\otimes$' may be reasonably expected to satisfy certain properties, such as $\forall C^\flat \in \mathcal{C}^\flat : \bot^\flat \otimes C^\flat = \bot^\flat$, these are not really required for what follows. The same applies to all the other operators we will introduce: only properties that are actually used will be singled out.

Parameter passing requires, roughly speaking, the ability of adding equality constraints to a constraint store. Notice that we assume $\mathcal{C}^\flat$ and its operations encode both the proper *constraint solver* and the so called *interface* between the *Herbrand engine* and the solver [16]. In particular, the interface is responsible for *type-checking* of the equations it receives. For example in CLP($\mathcal{R}$) the interface is responsible for the fact that $X = a$ cannot be consistently added to a constraint store where X was previously classified as numeric.

Another ingredient for defining the concrete semantics of any CLP system is the projection of a satisfiable constraint store onto a set of variables. This is modeled by the family of operators $\bigl\{\, \P^\flat_\Delta : \mathcal{C}^\flat \to \mathcal{C}^\flat \bigm| \Delta \in \wp_{\mathrm{f}}(\mathit{Vars}) \,\bigr\}$. If Δ is a finite set of variables and $C^\flat \in \mathcal{C}^\flat$ represents a satisfiable constraint store (i.e., $C^\flat \neq \bot^\flat$), then $\P^\flat_\Delta\, C^\flat$ represents the projection of $C^\flat$ onto the variables in Δ.

For each $\bar{s}, \bar{t} \in \mathbf{T}_{\mathbf{z}}^*$, we write $\varrho_{\bar{s}}(\bar{t})$ (read "rename $\bar{t}$ away from $\bar{s}$") to denote $\bar{t}[Z_{n+1}/Z_1, \ldots, Z_{n+m}/Z_m]$, where $n = \bigl|\mathit{vars}(\bar{s})\bigr|$ and $m = \bigl|\mathit{vars}(\bar{t})\bigr|$. The ϱ operator is useful for concatenating normalized term-tuples, still obtaining a normalized term-tuple, since we have $\bar{s} :: \varrho_{\bar{s}}(\bar{t}) \in \mathbf{T}_{\mathbf{z}}^{*}$. The renaming apart has to be extended to elements of $\mathcal{D}^{\flat}$. Let $C^{\flat} \in \mathcal{C}^{\flat}$ such that $FV(C^{\flat}) \subseteq vars(\bar{t})$. Then $\varrho_{\bar{s}}\bigl((\bar{t}, C^{\flat})\bigr)$ denotes the pair $(\varrho_{\bar{s}}(\bar{t}), C_1^{\flat})$, where $C_1^{\flat} \in \mathcal{C}^{\flat}$ is obtained from $C^{\flat}$ by applying the same renaming applied to $\bar{t}$ in order to obtain $\varrho_{\bar{s}}(\bar{t})$.

Term tuples are normalized by a *normalization function* $\eta \colon \mathcal{T}_{Vars}^{*} \to \mathbf{T}_{\mathbf{z}}^{*}$ such that, for each $\bar{u} \in \mathcal{T}_{Vars}^{*}$, the resulting tuple $\eta(\bar{u}) \in \mathbf{T}_{\mathbf{z}}^{*}$ is a variant of $\bar{u}$. As for ϱ, the normalization function has to be extended to elements of $\mathcal{D}^{\flat}$. Suppose that $G^{\flat} \in \mathcal{C}^{\flat}$ where $FV(G^{\flat}) \subseteq vars(\bar{u})$. then $\eta\bigl((\bar{u}, G^{\flat})\bigr)$ denotes $(\eta(\bar{u}), G_1^{\flat}) \in \mathcal{D}^{\flat}$ where it is assumed that $G_1^{\flat}$ can be obtained from $G^{\flat}$ by applying the same renaming applied to $\bar{u}$ in order to obtain $\eta(\bar{u})$.

We will now show how any abstract domain can be upgraded so as to capture structural information by means of the Pattern$(\cdot)$ construction. Then we will focus our attention on the abstract semantic operators.

4.1 From $\mathcal{D}^{\sharp}$ to Pattern$(\mathcal{D}^{\sharp})$

Since one of the driving aims of this work is maximum generality, we refer to a very weak abstract interpretation framework [12]. To start with, we assume very little on abstract domains.

Definition 2. (Abstract domain for $\mathcal{D}^{\flat}$.) *An* abstract domain for $\mathcal{D}^{\flat}$ *is a set $\mathcal{P}^{\sharp}$ equipped with a preorder relation '$\preceq$' $\subseteq \mathcal{P}^{\sharp} \times \mathcal{P}^{\sharp}$, an order preserving function $\gamma \colon \mathcal{P}^{\sharp} \to \mathcal{D}^{\flat}$, and a least element $\bot^{\sharp}$ such that $\gamma(\bot^{\sharp}) = \varnothing$. Moreover, γ is such that if $(\bar{p}_1, C^{\flat}) \in \gamma(E^{\sharp})$, and $\mathfrak{T} \models C^{\flat} \to \bar{p}_1 = \bar{p}_2$, then $\eta\bigl((\bar{p}_2, C^{\flat})\bigr) \in \gamma(E^{\sharp})$.*

Informally, $\mathcal{P}^{\sharp}$ is a set of abstract properties on which the notion of "relative precision" is captured by the preorder '$\preceq$'. Moreover, $\mathcal{P}^{\sharp}$ is related to the concrete domain $\mathcal{D}^{\flat}$ by means of a *concretization function* γ that specifies the soundness correspondence between $\mathcal{D}^{\flat}$ and $\mathcal{P}^{\sharp}$. The distinguished element $\bot^{\sharp}$ models an impossible state of affairs. In this framework, $d^{\sharp} \in \mathcal{P}^{\sharp}$ is a safe approximation of $d^{\flat} \in \mathcal{D}^{\flat}$ if and only if $d^{\flat} \subseteq \gamma(d^{\sharp})$.

Suppose we are given an abstract domain complying with Definition 2. Here is how it can be upgraded with explicit structural information.

Definition 3. (The Pattern$(\cdot)$ **construction.)** *Let $\mathcal{D}^{\sharp}$ be an abstract domain for $\mathcal{D}^{\flat}$ and let γ be its concretization function. Then*

$$\mathrm{Pattern}(\mathcal{D}^{\sharp}) = \{\bot_p^{\sharp}\} \cup \left\{ (\bar{s}, E^{\sharp}) \in \mathbf{T}_{\mathbf{z}}^{*} \times \mathcal{D}^{\sharp} \;\middle|\; \gamma(E^{\sharp}) \subseteq \mathbf{T}_{\mathbf{z}}^{|vars(\bar{s})|} \times \mathcal{C}^{\flat} \right\}.$$

The meaning of each element $(\bar{s}, E^{\sharp}) \in \mathrm{Pattern}(\mathcal{D}^{\sharp})$ is given by the concretization function $\gamma_p \colon \mathrm{Pattern}(\mathcal{D}^{\sharp}) \to \mathcal{D}^{\flat}$ such that $\gamma_p(\bot_p^{\sharp}) = \varnothing$ and

$$\gamma_p\bigl((\bar{s}, E^{\sharp})\bigr) = \left\{ \eta\bigl((\bar{r}, C^{\flat})\bigr) \;\middle|\; \begin{array}{l} (\bar{u}, C^{\flat}) \in \gamma(E^{\sharp}) \\ \mathfrak{T} \models C^{\flat} \to \bar{r} = \bar{s}'\bigl[\bar{u}/vseq(\bar{s}')\bigr] \end{array} \right\}.$$

We also define the binary relation '$\preceq_p$' $\subseteq \mathrm{Pattern}(\mathcal{D}^{\sharp}) \times \mathrm{Pattern}(\mathcal{D}^{\sharp})$ given, for each $d_1^{\sharp}, d_2^{\sharp} \in \mathrm{Pattern}(\mathcal{D}^{\sharp})$, by $d_1^{\sharp} \preceq_p d_2^{\sharp} \iff \gamma_p(d_1^{\sharp}) \subseteq \gamma_p(d_2^{\sharp})$.

It can be seen that Pattern($\mathcal{D}^\sharp$) is an abstract domain in the sense of Definition 2 provided $\mathcal{D}^\sharp$ is. Thus Pattern($\mathcal{D}^\sharp$) can constitute the basis for designing an abstract semantics for CLP. This will usually require selecting an abstract semantic function on Pattern($\mathcal{D}^\sharp$), an effective convergence criterion for the abstract iteration sequence (notice that the '$\preceq$' and '$\preceq_p$' relations are not required to be computable), and perhaps a convergence acceleration method ensuring rapid termination of the abstract interpreter [12]. The last ingredient to complete the recipe is a computable way to associate an abstract description $d^\sharp \in \mathrm{Pattern}(\mathcal{D}^\sharp)$ to each concrete property $d^\flat \in \mathcal{D}^\flat$. For this purpose, the existence of a computable function $\alpha_p \colon \mathcal{D}^\flat \to \mathrm{Pattern}(\mathcal{D}^\sharp)$ such that, for each $d^\flat \in \mathcal{D}^\flat$, $d^\flat \subseteq \gamma_p\bigl(\alpha_p(d^\flat)\bigr)$ is assumed.

While one option is to design an abstract semantics based on Pattern($\mathcal{D}^\sharp$) from scratch, it is more interesting to start with an abstract semantics centered around $\mathcal{D}^\sharp$. In this case, it is possible to systematically lift the semantic construction to Pattern($\mathcal{D}^\sharp$).

4.2 Operations over $\mathcal{D}^\sharp$ and Pattern($\mathcal{D}^\sharp$)

We now present the abstract operations we assume on $\mathcal{D}^\sharp$ and the derived operations over Pattern($\mathcal{D}^\sharp$). Each operator on $\mathcal{D}^\sharp$ is introduced by means of safety conditions that ensure the safety of the derived operators over Pattern($\mathcal{D}^\sharp$).

Given the abstract domain, there are still many degrees of freedom for the design of a constructive abstract semantics. Thus, choices have to be made in order to give a precise characterization. In what follows we continue to strive for maximum generality. Where this is not possible we detail the design choices we have made in the development of the China analyzer [1]. While some things may need adjustments for other analysis frameworks, the general principles should be clear enough for anyone to make the necessary changes.

Meet with Renaming Apart We call *meet with renaming apart* (denoted by '$\rhd$') the operation of taking two descriptions in $\mathcal{D}^\sharp$ and, roughly speaking, juxtaposing them. This is needed when "solving" a clause body with respect to the current interpretation and corresponds, at the concrete level, to a renaming followed by an application of the '$\otimes$' operator. Its counterpart on Pattern($\mathcal{D}^\sharp$) is denoted by 'rmeet' and defined as follows.

Definition 4. ('$\rhd$' and 'rmeet') *Let '$\rhd$': $\mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ be such that, for each $E_1^\sharp, E_2^\sharp \in \mathcal{D}^\sharp$,*

$$\gamma(E_1^\sharp \rhd E_2^\sharp) = \left\{ \eta\bigl((\bar{r}, C_1^\flat \otimes G_2^\flat)\bigr) \;\middle|\; \begin{array}{l} (\bar{r}_1, C_1^\flat) \in \gamma(E_1^\sharp) \\ (\bar{r}_2, C_2^\flat) \in \gamma(E_2^\sharp) \\ (\bar{w}_2, G_2^\flat) = \varrho_{\bar{r}_1}\bigl((\bar{r}_2, C_2^\flat)\bigr) \\ \mathfrak{T} \models (C_1^\flat \otimes G_2^\flat) \to \bar{r} = \bar{r}_1 :: \bar{w}_2 \end{array} \right\}.$$

Then, we define $\mathrm{rmeet}\bigl((\bar{s}_1, E_1^\sharp), (\bar{s}_2, E_2^\sharp)\bigr) = \bigl(\bar{s}_1 :: \varrho_{\bar{s}_1}(\bar{s}_2), E_1^\sharp \rhd E_2^\sharp\bigr)$, *for each* $(\bar{s}_1, E_1^\sharp), (\bar{s}_2, E_2^\sharp) \in \mathrm{Pattern}(\mathcal{D}^\sharp)$.

A consequence of this definition is that there is no precision loss in 'rmeet' [3].

Parameter Passing Concrete parameter passing is realized by an extended unification procedure. Unification is *extended* because it must involve the constraint solver(s). Remember that our notion of "constraint solver" includes also the interface between the Herbrand engine and the proper solver [16]. The interface needs to be notified about all the bindings performed by the Herbrand engine in order to maintain consistency between the solver and the Herbrand part. We also assume that CLP programs are normalized in such a way that interpreted function symbols only occur in explicit constraints (note that this is either required by the language syntax itself, as in the case of the clp(Q, R) libraries of SICStus Prolog, or is performed automatically by the CLP system).

At the abstract level we do not prescribe the use of any particular algorithm. This is to keep our approach as general as possible. For instance, an implementor is not forced to use any particular representation for term-tuples (as in [11]). Similarly, one can choose any sound unification procedure that works well with the selected representation. Of particular interest is the possibility of choosing a representation and procedure that closely match the ones employed in the concrete language being analyzed. In this case, all the easy steps typical of any unification procedure (functor name/arity checks, peeling, and so on) will be handled, at the abstract level, exactly as they are at the concrete level. The only crucial operation in abstract parameter passing over Pattern($\mathcal{D}^\sharp$) is the binding of an abstract variable to an abstract term. This is performed by first applying a non-cyclic approximation of the binding to the pattern component and then notifying the original (possibly cyclic) binding to the abstract constraint component. The correctness of this approach can be proved [3] by assuming the existence of a bind operator on the underlying abstract constraint system satisfying the following condition.

Definition 5. (bind) *Let $E^\sharp \in \mathcal{D}^\sharp$ be a description such that $\gamma(E^\sharp) \subseteq \mathbf{T}_{\mathbf{z}}^m \times \mathcal{C}^\flat$. Let $\bar{Z} = (Z_1, \dots, Z_m)$, $u \in \mathcal{T}_{\bar{Z}}$, $vseq(u) = (Z_{j_1}, \dots, Z_{j_l})$ and let $1 \leq h \leq m$. Then, define $(k_1, \dots, k_{m_1}) = \bigl((1, \dots, h-1) :: \bigl((j_1, \dots, j_l) \setminus \{1, \dots, h-1\}\bigr) :: \bigl((h+1, \dots, m) \setminus \{j_1, \dots, j_l\}\bigr)\bigr)$. If $E_1^\sharp = \mathrm{bind}(E^\sharp, u, Z_h)$, then,*

$$\gamma(E_1^\sharp) \supseteq \left\{ \eta\bigl((\bar{q}\theta, C_1^\flat)\bigr) \;\middle|\; \begin{array}{l} (\bar{p}, C^\flat) \in \gamma(E^\sharp) \\ \bar{p} = (p_1, \dots, p_m) \\ \bar{q} = (p_{k_1}, \dots, p_{k_{m_1}}) \\ \theta \textit{ is an idempotent substitution} \\ FV(C_1^\flat) \subseteq vars(\bar{q}\theta) \\ \mathfrak{T} \models \theta \leftarrow (Z'_h = u')[\bar{p}/\bar{Z}'] \\ \mathfrak{T} \models C_1^\flat \leftrightarrow \bigl((Z'_h = u')[\bar{p}/\bar{Z}'] \wedge C^\flat\bigr)\theta \end{array} \right\}.$$

Note that $m_1 = m - 1$ if $Z_h \notin vars(u)$, and $m_1 = m$, otherwise.

To motivate and explain the above condition on $E_1^\sharp = \mathrm{bind}(E^\sharp, u, Z_h)$, suppose that $\bar{p}$ is the pattern component and $C^\flat$ the constraint component of an element in the concretization of $E^\sharp$. Now, the pattern components of elements of the abstract domain $\mathrm{Pattern}(\mathcal{D}^\sharp)$ are always in normal form and thus, after applying the binding $[u'/Z'_h]$ to an element of $E^\sharp$, we must apply the normalization function so that the result is also in $\mathrm{Pattern}(\mathcal{D}^\sharp)$. This will first remove the h-th term Z_h in the case that Z_h does not occur in u and then permute the remaining elements of $\bar{Z}$. A corresponding operation is applied to the pattern $\bar{p}$. That is, $\bar{q}$ is constructed from $\bar{p}$ first by removing the h-th term p_h in the case that Z_h does not occur in u and then by applying the same permutation as before on the remaining elements of $\bar{p}$. As a most general solution ϕ to $(Z'_h = u')[\bar{p}/\bar{Z}']$ may be cyclic, only an approximation of ϕ, the idempotent substitution θ, is applied to $\bar{q}$. The actual solution ϕ together with $C^\flat\theta$ is captured by the constraint $C_1^\flat$. Finally, note that the new pattern component $\bar{q}\theta$ may not be in normal form, so that in the condition for bind it is the normalized variant of $(\bar{q}\theta, C_1^\flat)$ that must be in the concretization of $E_1^\sharp$.

We refer the reader to [3] for a description of how any correct unification algorithm can be transformed into a correct (abstract) unification algorithm for $\mathrm{Pattern}(\mathcal{D}^\sharp)$ using the bind operator and the normalization function η.

Projection When all the goals in a clause body have been solved, projection is used to restrict the abstract description to the tuple of arguments of the clause's head. The projection operations on $\mathcal{D}^\flat$ consist simply in dropping a suffix of the term-tuple component, with the consequent projection on the underlying constraint domain.

Definition 6. **('$\mathrm{project}_k^\flat$')** $\{\, \mathrm{project}_k^\flat \colon \mathcal{D}^\flat \to \mathcal{D}^\flat \mid k \in \mathbb{N} \,\}$ *is a family of operations such that, for each* $k \in \mathbb{N}$ *and each* $(\bar{u}, C^\flat) \in \mathcal{D}^\flat$ *with* $|\bar{u}| \geq k$*, if we define* $\Delta = \mathit{vars}\bigl(\mathrm{prefix}_k(\bar{u})\bigr)$*, then* $\mathrm{project}_k^\flat\bigl((\bar{u}, C^\flat)\bigr) = \bigl(\mathrm{prefix}_k(\bar{u}), \P_\Delta^\flat\, C^\flat\bigr)$*.*

We now introduce the corresponding projection operations on $\mathrm{Pattern}(\mathcal{D}^\sharp)$ and, in order to establish their correctness, we impose a safety condition on the projection operations of $\mathcal{D}^\sharp$.

Definition 7. **($\P_k^\sharp$ and $\mathrm{project}_k^\sharp$)** *Assume we are given a family of operations* $\{\, \P_k^\sharp \colon \mathcal{D}^\sharp \to \mathcal{D}^\sharp \mid k \in \mathbb{N} \,\}$ *such that, for each* $E^\sharp \in \mathcal{D}^\sharp$ *with* $\gamma(E^\sharp) \subseteq \mathbf{T}_{\mathbf{z}}^m \times \mathcal{C}^\flat$ *and each* $k \leq m$*,* $\gamma\bigl(\P_k^\sharp\, E^\sharp\bigr) \supseteq \{\, \mathrm{project}_k^\flat\bigl((\bar{u}, C^\flat)\bigr) \mid (\bar{u}, C^\flat) \in \gamma(E^\sharp) \,\}$*. Then, for each* $(\bar{s}, E^\sharp) \in \mathrm{Pattern}(\mathcal{D}^\sharp)$ *such that* $\bar{s} \in \mathbf{T}_{\mathbf{z}}^m$ *and each* $k \leq m$*, we define* $\mathrm{project}_k^\sharp\bigl((\bar{s}, E^\sharp)\bigr) = \bigl(\mathrm{prefix}_k(\bar{s}), \P_j^\sharp\, E^\sharp\bigr)$*, where* $j = \bigl|\mathit{vars}\bigl(\mathrm{prefix}_k(\bar{s})\bigr)\bigr|$*.*

With these definitions '$\mathrm{project}_k^\sharp$' is correct with respect to '$\mathrm{project}_k^\flat$' [3].

Remapping The operation of *remapping* is used to adapt a description in $\mathrm{Pattern}(\mathcal{D}^\sharp)$ to a different, less precise, pattern component. Remapping is essential to the definition of various *join* and *widening* operators. Consider a description $(\bar{s}, E_{\bar{s}}^\sharp) \in \mathrm{Pattern}(\mathcal{D}^\sharp)$ and a pattern $\bar{r}' \in \mathbf{T}_{\mathbf{z}'}^*$ such that $\bar{r}'$ is an anti-instance of $\bar{s}$. We want to obtain $E_{\bar{r}}^\sharp \in \mathcal{D}^\sharp$ such that $\gamma_p\bigl((\bar{r}, E_{\bar{r}}^\sharp)\bigr) \supseteq \gamma_p\bigl((\bar{s}, E_{\bar{s}}^\sharp)\bigr)$. This is what we call *remapping* $(\bar{s}, E_{\bar{s}}^\sharp)$ *to* $\bar{r}'$.

Definition 8. **('remap')** *Let* $(\bar{s}, E^\sharp_{\bar{s}}) \in \mathrm{Pattern}(\mathcal{D}^\sharp)$ *be a description with* $\bar{s} \in \mathbf{T}^k_{\mathbf{z}}$ *and let* $\bar{r}' \in \mathbf{T}^k_{\mathbf{z}'}$ *be an anti-instance of* $\bar{s}$*. Assume also* $|vars(\bar{r})| = m$ *and let* $\bar{u} \in \mathbf{T}^m_{\mathbf{z}}$ *be the unique tuple such that* $\bar{r}'[\bar{u}/vseq(\bar{r}')] = \bar{s}$*. Then the operation* $\mathrm{remap}(\bar{s}, E^\sharp_{\bar{s}}, \bar{r}')$ *yields* $E^\sharp_{\bar{r}}$ *such that* $\gamma(E^\sharp_{\bar{r}}) \supseteq \gamma_p\big((\bar{u}, E^\sharp_{\bar{s}})\big)$*.*

Observe that the remap function is closely related to the residual abstraction function α' of Figure 1.[1] With this definition, the specification of 'remap' meets our original requirement [3].

Upper Bound Operators A concrete (collecting) semantics for CLP will typically use set union to gather results coming from different computation paths. We assume that our base domain $\mathcal{D}^\sharp$ captures this operation by means of an upper bound operator '$\oplus$'. Namely, for each $E^\sharp_1, E^\sharp_2 \in \mathcal{D}^\sharp$ and each $i = 1,\ 2$, we have that $E^\sharp_i \preceq E^\sharp_1 \oplus E^\sharp_2$. This is used to merge descriptions arising from the different computation paths explored during the analysis.

The operation of merging two descriptions in $\mathrm{Pattern}(\mathcal{D}^\sharp)$ is defined in terms of 'remap'. Let $(\bar{s}_1, E^\sharp_1)$ and $(\bar{s}_2, E^\sharp_2)$ be two descriptions with $\bar{s}_1, \bar{s}_2 \in \mathbf{T}^k_{\mathbf{z}}$. The resulting description is $\big(\bar{r}, E^\sharp_1 \oplus E^\sharp_2\big)$, where $\bar{r}' \in \mathbf{T}^k_{\mathbf{z}'}$ is an anti-instance of both $\bar{s}_1$ and $\bar{s}_2$, and $E^\sharp_i = \mathrm{remap}(\bar{s}_i, E^\sharp_i, \bar{r}')$, for $i = 1,\ 2$. We note again that $\bar{r}'$ might be the least common anti-instance of $\bar{s}_1$ and $\bar{s}_2$, or it can be a further approximation of $\mathrm{lca}(\bar{s}_1, \bar{s}_2)$: this is one of the degrees of freedom of the framework. Thus, the family of operations we are about to present is parameterized with respect to a common anti-instance function and the analyzer may dynamically choose which anti-instance function is used at each step.

Definition 9. **('join_ϕ')** *Let* ϕ *be any common anti-instance function. The operation (partial function)* $\mathrm{join}_\phi \colon \wp_{\mathrm{f}}\big(\mathrm{Pattern}(\mathcal{D}^\sharp)\big) \rightarrowtail \mathrm{Pattern}(\mathcal{D}^\sharp)$ *is defined as follows. For each* $k \in \mathbb{N}$ *and each finite family* $F = \big\{ (\bar{s}_i, E^\sharp_i) \bigm| i \in I \big\}$ *of elements of* $\mathrm{Pattern}(\mathcal{D}^\sharp)$ *such that* $\bar{s}_i \in \mathbf{T}^k_{\mathbf{z}}$ *for each* $i \in I$*,* $\mathrm{join}_\phi(F) = (\bar{r}, E^\sharp)$*, where* $\bar{r}' = \phi\big(\{\, \bar{s}_i \mid i \in I \,\}\big)$ *and* $E^\sharp = \bigoplus_{i \in I} \mathrm{remap}(\bar{s}_i, E^\sharp_i, \bar{r}')$*.*

If ϕ is any common anti-instance function then 'join_ϕ' is an upper bound operator [3].

Widenings It is possible to devise a (completely unnatural) abstract domain $\mathcal{D}^\sharp$ that enjoys the *ascending chain condition*[2] still preventing $\mathrm{Pattern}(\mathcal{D}^\sharp)$ from possessing the same property. This despite the fact that any element of $\mathbf{T}^n_{\mathbf{z}}$ has a finite number of distinct anti-instances in $\mathbf{T}^n_{\mathbf{z}'}$. However, this problem is of no practical interest if the analysis applies 'join_ϕ' at each step of the iteration sequence. In this case, if we denote by $(\bar{s}_j, E^\sharp_j) \in \mathrm{Pattern}(\mathcal{D}^\sharp)$ the description at step $j \in \mathbb{N}$, we have $(\bar{s}_{i+1}, E^\sharp_{i+1}) = \mathrm{join}_\phi\big(\{(\bar{s}_i, E^\sharp_i), \ldots\}\big)$, assuming no widening

[1] Indeed, one can define $\alpha' = \lambda(\bar{s}, E^\sharp) \in \mathbf{T}^k_{\mathbf{z}} \times \mathcal{D}^\sharp \,.\, \mathrm{remap}\big(\bar{s}, E^\sharp, (Z'_1, \ldots, Z'_k)\big)$.

[2] Namely, each strictly increasing chain is finite.

is employed. This implies that $\bar{s}'_{i+1}$ is an anti-instance of $\bar{s}_i$. As any ascending chain in $\mathbf{T}_{\mathbf{z}}^n$ is finite, the iteration sequence will eventually stabilize if $\mathcal{D}^\sharp$ enjoys the ascending chain condition.

In some cases, however, rapid termination of the analysis on $\mathcal{D}^\sharp$ can only be ensured by using one or more widening operators $\nabla \colon \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ [13]. These can be lifted to work on $\mathrm{Pattern}(\mathcal{D}^\sharp)$. As an example, we show the default lifting used by the CHINA analyzer:

$$\mathrm{widen}\bigl((\bar{s}_1, E_1^\sharp), (\bar{s}_2, E_2^\sharp)\bigr) = \begin{cases} (\bar{s}_2, E_2^\sharp), & \text{if } \bar{s}_1 \neq \bar{s}_2; \\ (\bar{s}_2, E_1^\sharp \nabla E_2^\sharp), & \text{if } \bar{s}_1 = \bar{s}_2. \end{cases} \tag{1}$$

This operator refrains from widening unless the pattern component has stabilized. A more drastic choice for a widening is given by

$$\mathrm{Widen}\bigl((\bar{s}_1, E_1^\sharp), (\bar{s}_2, E_2^\sharp)\bigr) = \bigl(\bar{s}_2, \mathrm{remap}(\bar{s}_1, E_1^\sharp, \bar{s}'_2) \nabla E_2^\sharp\bigr). \tag{2}$$

Widening operators only need to be evaluated over $(\bar{s}_1, E_1^\sharp)$ and $(\bar{s}_2, E_2^\sharp)$ when $\bar{s}'_2$ is an anti-instance of $\bar{s}_1$. Thus, as $\mathbf{T}_{\mathbf{z}}^n$ satisfies the ascending chain condition, 'widen' and 'Widen' are well-defined widening operators on $\mathrm{Pattern}(\mathcal{D}^\sharp)$ [3].

Besides ensuring termination, widening operators are also used to accelerate convergence of the analysis. It is therefore important to be able to define widening operators on $\mathrm{Pattern}(\mathcal{D}^\sharp)$ without relying on the existence of corresponding widenings on $\mathcal{D}^\sharp$. There are many possibilities in this direction and some of them are currently under experimental evaluation. Just note that any upper bound operator 'join_ϕ' can be regarded as a widening as soon as the common anti-instance function ϕ is different from the lca. In order to ensure the convergence of the abstract computation, we will only consider widening operators on $\mathrm{Pattern}(\mathcal{D}^\sharp)$ satisfying the following (very reasonable) condition: if $(\bar{s}, E^\sharp)$ is the result of the widening applied to $(\bar{s}_1, E_1^\sharp)$ and $(\bar{s}_2, E_2^\sharp)$, where $\bar{s}'_2$ is an anti-instance of $\bar{s}_1$, then $\bar{s}'$ is an anti-instance of $\bar{s}_2$. Both widen and Widen comply with this restriction.

Comparing Descriptions The *comparison* operation on $\mathrm{Pattern}(\mathcal{D}^\sharp)$ is used by the analyzer in order to check whether a local fixpoint has been reached.

Definition 10. **('compare')** *Let '$\precsim$' $\subseteq \mathcal{D}^\sharp \times \mathcal{D}^\sharp$ be a computable preorder that correctly approximates '$\preceq$', that is, for each $E_1^\sharp, E_2^\sharp \in \mathcal{D}^\sharp$, we have $E_1^\sharp \preceq E_2^\sharp$ whenever $E_1^\sharp \precsim E_2^\sharp$. The* approximated ordering relation over $\mathrm{Pattern}(\mathcal{D}^\sharp)$, *denoted by* 'compare' $\subseteq \mathrm{Pattern}(\mathcal{D}^\sharp) \times \mathrm{Pattern}(\mathcal{D}^\sharp)$, *is defined, for each* $(\bar{s}_1, E_1^\sharp), (\bar{s}_2, E_2^\sharp) \in \mathrm{Pattern}(\mathcal{D}^\sharp)$, *by* $\mathrm{compare}\bigl((\bar{s}_1, E_1^\sharp), (\bar{s}_2, E_2^\sharp)\bigr) \iff \bigl(\bar{s}_1 = \bar{s}_2 \land E_1^\sharp \precsim E_2^\sharp\bigr)$.

It must be stressed that the above ordering is "approximate" since it does not take into account the peculiarities of $\mathcal{D}^\sharp$. More refined orderings can be obtained in a domain-dependent way, namely, when $\mathcal{D}^\sharp$ has been fixed. It is easy to show that compare is a preorder over $\mathrm{Pattern}(\mathcal{D}^\sharp)$ that correctly approximates the approximation ordering '$\preceq_p$' [3]. The ability of comparing descriptions only when they have the same pattern is not restrictive in our setting. Indeed, the definition of join_ϕ and the condition we imposed on widenings ensure that any two descriptions arising from consecutive steps of the iteration sequence are ordered by the anti-instance relation. When combined with the ascending chain condition of the pattern component, this allows to inherit termination from the underlying domain $\mathcal{D}^\sharp$.

5 Experimental Evaluation

We have conducted an extensive experimentation on the analysis using the Pattern($\cdot$) construction: this allowed us to tune the implementation and gain insight on the implications of keeping track of explicit structural information. To put ourselves in a realistic situation, we assessed the impact of the Pattern($\cdot$) construction on *Modes*, a very precise and complex domain for mode analysis. This captures information on simple types, groundness, boundedness, pair-sharing, freeness, and linearity. It is a combination of, among other things, two copies of the GER representation for *Pos* [5] — one for groundness and one for boundedness — and the non-redundant pair-sharing domain *PSD* [4] with widening as described in [22]. Each of these domains has been suitably extended to ensure correctness and precision of the analysis even for systems that omit the occur-check [1, 15]. Some details on how the domains are combined can be found in [6].

The benchmark suite used for the development and tuning of the CHINA analyzer is probably the largest one ever employed for this purpose. The suite comprises all the programs we have access to (i.e., everything we could find by systematically dredging the Internet): 300 programs, 16 MB of code, 500 K lines, the largest program containing 10063 clauses in 45658 lines of code.

The comparison between *Modes* and Pattern(*Modes*) involves the two usual things: *precision* and *efficiency*. However, how are we going to compare the precision of the domain with explicit structural information with one without it? That is something that should be established in advance. Let us consider a simple but not trivial Prolog program: `mastermind.pl`.[3] Consider also the only direct query for which it has been written, '`?- play.`', and focus the attention on the procedure `extend_code/1`. A standard goal-dependent analysis of the program with the *Modes* domain is only able to tell something like

```
extend_code(A) :- list(A).
```

This means: "during any execution of the program, whenever `extend_code/1` succeeds it will have its argument bound to a *list cell* (i.e., a term whose principal functor is either `'.'/2` or `[]/0`)". Not much indeed. Especially because this can be established instantly by visual inspection: `extend_code/1` is *always* called with a list argument and this completes the proof. If we perform the analysis with Pattern(*Modes*) the situation changes radically. Here is what such a domain allows CHINA to derive:[4]

[3] Available at `http://www.cs.unipr.it/China/Benchmarks/Prolog/mastermind.pl`.
[4] Some extra groundness information obtained by the analysis has been omitted.

$x = \%\text{inc.}$	indep		ground		linear		free		bound	
	GI	GD	GI	GD	GI	GD	GI	GD	GI	GD
$x < 0$	0	1	0	0	0	1	0	0	0	0
$x = 0$	222	211	228	223	213	205	244	245	230	220
$0 < x \leq 2$	36	35	24	26	46	44	26	21	49	45
$2 < x \leq 5$	22	27	17	17	17	18	11	13	9	15
$5 < x \leq 10$	7	8	10	11	9	11	10	8	9	8
$x \geq 10$	13	18	21	23	15	21	9	13	3	12

Table 1. A summary of the *Modes* precision gained using structural information.

```
extend_code([([A|B],C,D)|E]) :- list(B), list(E),
  (functor(C,_,1);integer(C)), (functor(D,_,1);integer(D)),
  ground([C,D]), may_share([[A,B,E]]).
```

Under the circumstances mentioned above, this means: "the argument of procedure `extend_code/1` will be bound to a term of the form `[([A|B],C,D)|E]`, where `B` and `E` are bound to list cells; `C` is either bound to a functor of arity 1 or to an integer, and likewise for `D`; both `C` and `D` are ground, and (consequently) pair-sharing may only occur between `A`, `B`, and `E`".

It is clear that the analysis with Pattern($Modes$) yields much more information. However, it is not clear at all how to define a fair measure for this precision gain. The approach we have chosen is simple though unsatisfactory: throw away all the structural information at the end of the analysis and compare the usual numbers (i.e., number of ground variables, number of free variables and so on). With reference to the above example, this metric pretends that explicit structural information gives no precision improvements on the analysis of `extend_code/1` in `mastermind.pl`. In fact, once all the structural information has been discarded, the analysis with Pattern($Modes$) only specifies that, upon success, the argument of `extend_code/1` will be a list cell. In other words, we are measuring how the explicit structural information present in Pattern($Modes$) improves the precision on *Modes* itself, which is only a tiny part of the real gain in accuracy. The value of this extra precision can only be measured from the point of view of the target application of the analysis.

It is important to note that the experimental results we are about to report have been obtained without using any widening on the pattern component. The widening operations are only propagated to the underlying *Modes* domain by means of the 'widen' operator given in Eq. (1). Moreover, the merge operation employed is always 'join_{lca}'. For space limitations, here we can only summarize the results of the experimentation. The interested reader can find all the details at `http://www.cs.unipr.it/China`. As far as precision is concerned, we measure five different quantities: the total number of independent argument pairs (*indep*); the total number of *ground* argument positions; the total number of *linear* argument positions; the total number of *free* argument positions; and the total number of *bound* (or *nonvar*) argument positions.

time difference in seconds	# prog.		% prog.	
	GI	GD	GI	GD
degradation ≥ 1	9	20	100.0	100.0
$0.5 \leq$ degradation < 1	2	4	97.0	93.3
$0.2 \leq$ degradation < 0.5	15	18	96.3	92.0
degradation < 0.2	105	106	91.3	86.0
same time	90	77	56.3	50.7
improvement < 0.2	34	31	26.3	25.0
$0.2 \leq$ improvement < 0.5	11	11	15.0	14.7
$0.5 \leq$ improvement < 1	9	5	11.3	11.0
improvement ≥ 1	25	28	8.3	9.3

Table 2. A summary on efficiency: the distribution of analysis time differences.

Since we are completely disregarding the precision gains coming from structural information in itself, our results give a (very pessimistic) lower bound on the overall precision improvement. The results are summarized by partitioning the benchmark suite into six classes of programs, identified by the percentage increase in precision due to the Pattern($\cdot$) construction. Table 1 gives the cardinalities of these classes for both goal-independent (GI) and goal-dependent (GD) analyses. A precision increase, on at least one of the measured quantities, is observed on more than one third of the benchmarks. The only precision decrease is due to the interaction between the Pattern($\cdot$) construction and the widenings used in the *Modes* domain. It is also worth observing that, on average, goal-dependent analysis is more likely to benefit from the addition of structural information.

In order to evaluate the impact on efficiency of the Pattern($\cdot$) transformation we computed the fixpoint evaluation time for all the programs, both with the *Modes* and with the Pattern(*Modes*) domains. Results are summarized by partitioning the benchmark suite into a number of classes and giving the cardinality of each class. As a first parameter, we considered the absolute time difference observed for each program.[5] Table 2 gives the cardinality of 9 classes, distinguishing between GI and GD analyses. The numbers show that the full range of possible behaviors is indeed observable. Quite surprisingly, it is not uncommon, although inherently more precise and complex, for the case with the Pattern($\cdot$) construction to result in significant time improvements. The reason for this is only partly due to the enhanced ability of the Pattern component to be able to detect and hence prune failed computation paths. Most importantly, the description of a set of tuples of terms in Pattern(*Modes*) is often much more efficient

[5] As the benchmark suite comprises several real programs of very respectable size, we believe that absolute time comparison is what really matters to assess the feasibility of the Pattern($\cdot$) construction with respect to the underlying domain. A time difference less than one second is an approximation of "the user will not notice." The experiments were conducted on a PC equipped with an AMD Athlon clocked at 700 MHz, 256 MB of RAM, and running Linux 2.2.16.

T = time in secs.	GI			GD		
	w/o SI	w SI	diff.	w/o SI	w SI	diff.
$T \geq 10$	15	11	-4	22	17	-5
$5 \leq T < 10$	9	5	-4	10	11	$+1$
$1 \leq T < 5$	32	35	$+3$	35	43	$+8$
$0.5 \leq T < 1$	11	21	$+10$	20	23	$+3$
$0.2 \leq T < 0.5$	27	38	$+11$	37	46	$+9$
$0.1 \leq T < 0.2$	164	158	-6	155	146	-9
$T < 0.1$	42	32	-10	21	14	-7

Table 3. A summary on efficiency: the distribution of analysis times.

than the corresponding description in *Modes*. Percentages in the columns on the right show how many programs are at least as good as the corresponding class. For instance, more than 85% of the benchmarks either reduce the analysis time or increase it by at most 0.2 secs. Since the occasional bad-behaving cases can be dealt with by defining a suitable widening operator on the pattern component, these results disprove the common belief that structural information has a heavy impact on the efficiency of the analysis.

As a second criterion, Table 3 partitions the benchmark suite into 7 classes based on their total fixpoint computation time, again distinguishing between GI and GD analysis. The columns labeled 'diff.' show how each class grows or shrinks because of the addition of structural information. It can be seen that the Pattern($\cdot$) construction causes only a minor change to the distribution, decreasing the number of benchmarks in both the fastest and the slowest classes.

6 Related Work

The use of explicit structural information has also been studied in [7], where abstract equation systems are integrated into an analysis domain tracking set-sharing, freeness, linearity and compoundness. While allowing for an implementation independent definition, this proposal still assumes the occur-check, therefore resulting in an unsound analysis for implemented CLP languages. An experimental evaluation on a small benchmark suite (19 programs) was reported by Mulkers et al. in [19, 20]. Here the investigation mainly focused on the comparison between different instances of the underlying domain, showing the positive impact of freeness and linearity information on both the precision and performance of the classical set-sharing analysis. The experiments on the integration of structural information, by means of a depth-k abstraction (replacing all subterms occurring at a depth greater or equal to k with fresh abstract variables) for values of k between 0 and 3, showed that the domain they employed was not suitable to the analysis of real programs and, in fact, even the analysis of a modest-sized program like 'ann' could only be carried out with depth-0 abstraction (i.e., with no structural information at all).

In [8], an alternative technique is proposed for augmenting a data-flow analysis with structural information. Instead of upgrading the analysis domain, this technique relies on program transformations. In this approach, called *untupling*, the data-flow analysis of a given program would be performed in four distinct phases. This new analysis technique is advocated for its simplicity and efficiency. Comparing their limited experimental evaluation to the one conducted in [11], the authors of [8] claim that the untupling approach is inherently more efficient than abstract domain enhancement. Our new performance results suggest that this conclusion may need reconsidering. On the other hand, the proposal in [8] may be simpler to implement despite the four phases required, especially if one has to start *from scratch*. However, the $\mathrm{Pattern}(\cdot)$ construction, besides being more precise and particularly efficient, is already implemented and has been thoroughly tested on a large number of benchmarks using the very expressive abstract domain *Modes*. Furthermore, as the implementation is in the form of a C++ template, only a very limited effort is required to upgrade any other abstract domain with structural information.

7 Conclusion

We have presented the rational construction of a generic domain for structural analysis of real CLP languages: $\mathrm{Pattern}(\mathcal{D}^\sharp)$, where the parameter $\mathcal{D}^\sharp$ is an abstract domain satisfying certain properties. We build on the parameterized $\mathtt{Pat}(\Re)$ domain of Cortesi et al. [10, 11], which is restricted to logic programs and requires the occur-check to be performed. However, while $\mathtt{Pat}(\Re)$ is presented as a *specific implementation* of a generic structural domain, our formalization is implementation-independent. Reasoning at a higher level of abstraction we are able to appeal to familiar notions of unification theory, while leaving considerable more latitude for the implementer. Indeed our results show that, contrary to popular belief, an analyzer incorporating structural information analysis based on our approach can be highly competitive even from the efficiency point of view.

References

1. R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy, March 1997.
2. R. Bagnara. A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages. *Science of Computer Programming*, 30(1–2):119–155, 1998.
3. R. Bagnara, P. M. Hill, and E. Zaffanella. Efficient structural information analysis for real CLP languages. Quaderno 229, Dipartimento di Matematica, Università di Parma, 2000. Available at `http://www.cs.unipr.it/~bagnara`.
4. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 2000. To appear.
5. R. Bagnara and P. Schachte. Factorizing equivalent variable pairs in ROBDD-based implementations of *Pos*. In A. M. Haeberer, editor, *Proc. of the "7th Int'l Conf. on Algebraic Methodology and Software Technology"*, vol. 1548 of *Lecture Notes in Computer Science*, pages 471–485, Amazonia, Brazil, 1999. Springer-Verlag, Berlin.

6. R. Bagnara, E. Zaffanella, and P. M. Hill. Enhancing Sharing for precision. In M. C. Meo and M. Vilares Ferro, editors, *Proc. of the "AGP'99 Joint Conf. on Declarative Programming"*, pages 213–227, L'Aquila, Italy, 1999.
7. M. Bruynooghe, M. Codish, and A. Mulkers. Abstract unification for a composite domain deriving sharing and freeness properties of program variables. In F. S. de Boer and M. Gabbrielli, editors, *Verification and Analysis of Logic Languages, Proc. of the W2 Post-Conference Workshop, Int'l Conf. on Logic Programming*, pages 213–230, Santa Margherita Ligure, Italy, 1994.
8. M. Codish, K. Marriott, and C. Taboch. Improving program analyses by structure untupling. *Journal of Logic Programming*, 43(3):251–263, 2000.
9. A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming, APIC Studies in Data Processing*, vol. 16, pages 231–251. Academic Press, New York, 1982.
10. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Conceptual and software support for abstract domain design: Generic structural domain and open product. Technical Report CS-93-13, Brown University, Providence, RI, 1993.
11. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 38(1–3), 2000.
12. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
13. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. of the 4th Int'l Symp. on Programming Language Implementation and Logic Programming*, vol. 631 of *Lecture Notes in Computer Science*, pages 269–295, Leuven, Belgium, 1992. Springer-Verlag, Berlin.
14. R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *Journal of Logic Programming*, 25(3):191–247, 1995.
15. P. M. Hill, R. Bagnara, and E. Zaffanella. The correctness of set-sharing. In G. Levi, editor, *Static Analysis: Proc. of the 5th Int'l Symp.*, vol. 1503 of *Lecture Notes in Computer Science*, pages 99–114, Pisa, Italy, 1998. Springer-Verlag, Berlin.
16. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19&20:503–582, 1994.
17. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
18. J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.
19. A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. Report CW 198, Department of Computer Science, K. U. Leuven, Belgium, 1994.
20. A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. In L. Sterling, editor, *Logic Programming: Proc. of the 12th Int'l Conf. on Logic Programming*, MIT Press Series in Logic Programming, pages 781–795, Kanagawa, Japan, 1995. The MIT Press.
21. K. Musumbu. *Interprétation Abstraite des Programmes Prolog*. PhD thesis, Institut d'Informatique, Facultés Univ. Notre-Dame de la Paix, Namur, Belgium, 1990.
22. E. Zaffanella, R. Bagnara, and P. M. Hill. Widening Sharing. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, vol. 1702 of *Lecture Notes in Computer Science*, pages 414–431, Paris, 1999. Springer-Verlag, Berlin.