Converting One Type-Based Abstract Domain to Another

Gallagher, John Patrick; Puebla, German; Albert, Elvira

Published in:
Lecture Notes in Computer Science

Publication date:
2006

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Gallagher, J. P., Puebla, G., & Albert, E. (2006). Converting One Type-Based Abstract Domain to Another. *Lecture Notes in Computer Science*, 147-162.

Converting One Type-Based Abstract Domain to Another

John P. Gallagher[1], Germán Puebla[2], and Elvira Albert[3]

[1] Department of Computer Science, Univ. of Roskilde
jpg@ruc.dk
[2] School of Computer Science, Technical Univ. of Madrid
german@fi.upm.es
[3] School of Computer Science, Complutense Univ. of Madrid
elvira@sip.ucm.es

Abstract. The specific problem that motivates this paper is how to obtain abstract descriptions of the meanings of imported predicates (such as built-ins) that can be used when analysing a module of a logic program with respect to some abstract domain. We assume that abstract descriptions of the imported predicates are available in terms of some "standard" assertions. The first task is to define an abstract domain corresponding to the assertions for a given module and express the descriptions as objects in that domain. Following that they are automatically transformed into the analysis domain of interest. We develop a method which has been applied in order to generate call and success patterns from the `CiaoPP` assertions for built-ins, for any given regular type-based domain. In the paper we present the method as an instance of the more general problem of mapping elements of one abstract domain to another, with as little loss in precision as possible.

1 Motivation

When performing static analysis of a logic program, the source code for some parts of it may be inaccessible for some reason (the code might be in external modules, built-in system predicates, foreign-language libraries, and so on). In order to analyse such a program accurately, abstract descriptions of the behaviour of the missing code have to be supplied, otherwise some coarse over-approximation (or sometimes under-approximation) has to be used.

It can take considerable effort to specify the properties of built-ins and library predicates over a given abstract domain, and those properties need to be specified for each domain for which the calling code is to be analysed. Our intention is to specify once and for all the properties of library predicates, using a general and expressive abstract domain of descriptions; these specifications are then converted to another abstract domain when a particular analysis is to be performed.

The following general principles of abstract domain construction [1] are applied. Given two abstract interpretations of a concrete semantics, say $\mathcal{A}_1$ and

P.M. Hill (Ed.): LOPSTR 2005, LNCS 3901, pp. 147–162, 2006.

$\mathcal{A}_2$, with abstraction and concretisation functions α_1, γ_1, α_2 and γ_2 respectively, the aim is to translate descriptions in $\mathcal{A}_1$ to descriptions in $\mathcal{A}_2$. The best representative of an element $a_1 \in \mathcal{A}_1$ in $\mathcal{A}_2$ is $\alpha_2(\gamma_1(a_1))$. If we can implement a function equivalent to this we can just apply it to descriptions expressed using elements of $\mathcal{A}_1$ to obtain descriptions in $\mathcal{A}_2$. For the abstract domains that we consider, namely those based on regular types, we show that the function can be implemented by constructing the *reduced product domain* $\mathcal{A}_1 \star \mathcal{A}_2$, with concretisation function $\gamma_\star$. Our method can be presented as the computation, for a given element $a_1 \in \mathcal{A}_1$, of the corresponding element $a_\star \in \mathcal{A}_1 \star \mathcal{A}_2$, such that $\gamma_\star(a_\star) = \gamma_1(a_1)$, and then computing the most precise element a_2 in the domain of $\mathcal{A}_2$ such that $\gamma_\star(a_\star) \sqsubseteq \gamma_2(a_2)$. a_1 has an exact representative in $\mathcal{A}_1 \star \mathcal{A}_2$ but we cannot in general find an exact representative of $a_\star$ in $\mathcal{A}_2$. In our method, $\mathcal{A}_1$ is the general-purpose domain, while $\mathcal{A}_2$ is a particular analysis domain.

Assertions in `CiaoPP`*.* In the `CiaoPP` system [2] an assertion language is provided that allows properties of predicates to be stated in a flexible, general language. The properties of built-ins and many library predicates have been expressed in this assertion language. The question addressed in this work is how to use such information in an analysis over a (new) particular abstract domain. The method described in this paper allows us to take any given assertions about a module's imported predicates and translate them safely (and accurately) into the domain under consideration. We use domains based on regular types, realised as pre-interpretations [3], using a subset of the `CiaoPP` assertion language to explain the approach.

In the `CiaoPP` assertion language, approximations of the success set of a predicate can be specified, among many other aspects of computation. We do not enter into the notation for the assertions here; detailed examples can be seen in [2]. We can extract this information in the form of a set of abstract atoms of the form $p(d_1, \ldots, d_n)$, for a predicate p/n, where $d_1, \ldots, d_n$ are the names of abstract term descriptions defined within the `CiaoPP` system.

Example 1. The success of $length/2$ is described by $\{length(list, int)\}$. Here int is a primitive type and $list$ is defined by a set of regular type rules. If the analysis of interest concerns modes g (*ground*) and $nong$ (*non-ground*), then this description would be transformed automatically into $\{length(g, g), length(nong, g)\}$. In order to achieve this transformation we need to derive the information that a $list$ can be either ground or non-ground, while an int is ground.

An alternative approach (currently pursued in the `CiaoPP` system) is to define relationships between analysis domains in advance (a type lattice) [2]. For example, the fact that arithmetic expressions are ground can be pre-defined. Once that is done, an assertion, say, that the predicate $< /2$ succeeds with both arguments bound to arithmetic expressions can safely be translated into the modes domain as an assertion that both arguments are ground.

In contrast, the approach defined here allows arbitrary relationships to be derived automatically, for user-defined types as well as pre-defined ones. We

focus here on transforming assertions about success of predicates, but the same approach can be followed for assertions on calls.

Related Work. The most closely related work is concerned with systematically constructing abstract domains from other domains [1, 4]. These principles have been applied in combining different abstractions from primitive operations in the ASTRÉE analyser (see e.g [5]). We make use of the reduced product in this paper, but domain construction is not our main aim, but rather to transfer information from one given domain to another given domain. Although the principles are well understood (see Section 2), we do not know of other work that applies them systematically to this problem.

Section 2 explains the general principles behind our solution. In Section 3 we review the kind of abstract domain that we deal with, namely, domains based on regular types and define a general solution for such domains. In Section 4 we describe how we construct a single set of regular types from the various different kinds of assertion in the `CiaoPP` assertion language, and thus define the standard domain. Section 5 presents the procedure for mapping descriptions in the standard domain into any given user-supplied domain based on regular types. The soundness and precision of the procedure are established by relating it to the general solution. Section 6 contains the results of some experiments in transforming `CiaoPP` assertions into various simple mode and type domains. Finally in Section 7 we present conclusions and future work.

2 General Characterisation of the Problem

We restrict our attention to abstract interpretations based on a Galois connection, which is given by a 4-tuple $\langle(\mathcal{D}, \sqsubseteq_{\mathcal{D}}), (\mathcal{A}, \sqsubseteq_{\mathcal{A}}), \alpha, \gamma\rangle$ where $(\mathcal{D}, \sqsubseteq_{\mathcal{D}})$ and $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ are partially ordered sets, the concrete and abstract domain of interpretation respectively, and $\alpha : \mathcal{D} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathcal{D}$ are adjoined functions satisfying

$$\forall x \in \mathcal{D}, y \in \mathcal{A} : (\alpha(x) \sqsubseteq_{\mathcal{A}} y) \Longleftrightarrow (x \sqsubseteq_{\mathcal{D}} \gamma(y)).$$

$\alpha(x)$ represents the best possible description of some concrete object x in the abstract domain $\mathcal{A}$, while $\gamma(y)$ represents the most imprecise element of the concrete domain $\mathcal{D}$ that is described by some abstract object y.

If $(\mathcal{D}, \sqsubseteq_{\mathcal{D}})$ and $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ are complete lattices, then the functions α and γ determine each other; in particular we have $\alpha(x) = \sqcap \{y \mid x \sqsubseteq_{\mathcal{D}} \gamma(y)\}$, where $\sqcap$ is the meet operator in $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$.

Suppose $\langle(\mathcal{D}, \sqsubseteq_{\mathcal{D}}), (\mathcal{A}_1, \sqsubseteq_{\mathcal{A}_1}), \alpha_1, \gamma_1\rangle$ and $\langle(\mathcal{D}, \sqsubseteq_{\mathcal{D}}), (\mathcal{A}_2, \sqsubseteq_{\mathcal{A}_2}), \alpha_2, \gamma_2\rangle$ are two abstract interpretations with same concrete domain. Then given an element $a_1 \in \mathcal{A}_1$, we can compute the best representation in $\mathcal{A}_2$ of a_1 as $\alpha_2(\gamma_1(a_1))$.

In the applications considered in this paper, the domains are complete lattices. We are not provided explicitly with the abstraction function α. Therefore, the expression $\alpha_2(\gamma_1(a_1))$ mentioned above is rewritten as

$$\alpha_2(\gamma_1(a_1)) = \sqcap \{y \in \mathcal{A}_2 \mid \gamma_1(a_1) \sqsubseteq_{\mathcal{D}} \gamma_2(y)\}$$

This expression suggests that all elements y of the set $\mathcal{A}_2$ have to be enumerated, which is not practical in general. A practical algorithm for computing the required element of $\mathcal{A}_2$ is obtained in the remainder of the paper, for domains that are based on pre-interpretations.

3 Analysis Domains Based on Regular Types

As shown in [3], any set of regular types (a non-deterministic finite tree automaton) over a logic program's signature can be used to build a pre-interpretation, and hence an abstract interpretation of the program. In this section we summarise this family of abstract interpretations and some key properties.

A set of *regular types* is defined by a set of type symbols Q, a signature Σ, and a set of rules of the form $f(d_1, \ldots, d_n) \rightarrow d$, where $f/n \in \Sigma$ and $d, d_1, \ldots, d_n \in Q$. A set of regular type definitions can be seen as a finite tree automaton (FTA). For our purposes we regard the two notions as interchangeable, and speak of the states of an FTA as "types". (We assume that every state of an FTA is an accepting (or final) state.) Let Term_Σ be the set of terms constructible from the function symbols in Σ. Given a state (type) d, let $L(d) \subseteq \mathsf{Term}_\Sigma$ be the set of terms accepted by d; that is, for all $t \in L(d)$ there is a bottom-up derivation starting at t and ending at d. We can also think of $L(d)$ as standing for the terms of "type" d. Full details of these concepts can be found in the literature [6].

It is known [6] that an arbitrary FTA can be transformed to an equivalent *bottom-up deterministic* FTA (or DFTA). The defining condition of a DFTA is that there are no two rules with the same left hand side. An arbitrary FTA can also be *completed*, meaning that it is extended so that there exists a rule $f(d_1, \ldots, d_n) \rightarrow d$ for each choice of $f, d_1, \ldots, d_n$. (An extra state may need to be added to the FTA.) Let $\mathcal{Q}$ be the set of states of a complete DFTA. Thus $\{L(d) \mid d \in \mathcal{Q}\}$ is a disjoint partition of Term_Σ. That is, each $t \in \mathsf{Term}_\Sigma$ is accepted by exactly one state in a bottom-up derivation in a complete DFTA.

Example 2. Let $\Sigma = \{[]/0, [.|.]/2, a/0, s/1\}$. The following rules define a complete DFTA over Σ:

$$\{[] \rightarrow list, [list|list] \rightarrow list, [nonlist|list] \rightarrow list,\\ [list|nonlist] \rightarrow nonlist, [nonlist|nonlist] \rightarrow nonlist, a \rightarrow nonlist,\\ s(list) \rightarrow nonlist, s(nonlist) \rightarrow nonlist\}$$

The rules define two types *list* and *nonlist*. Each term in Term_Σ is accepted by one of these two. This induces a partition of Term_Σ into two disjoint sets, lists and non-lists. The above DFTA could be obtained by determinizing and completing the following FTA:

$$\{[] \rightarrow list, [dynamic|list] \rightarrow list,\ [] \rightarrow dynamic,\\ [dynamic|dynamic] \rightarrow dynamic, s(dynamic) \rightarrow dynamic, a \rightarrow dynamic\}$$

Note that the two types *list* and *dynamic* are not disjoint, in fact $L(list) \subset L(dynamic)$ in this case.

Representation of States in a Determinized FTA. Let Q be the set of states of an FTA. The textbook algorithm for determinization [6] constructs a DFTA whose set of states is some subset of 2^Q, say $\mathcal{Q}$. Let $\{d_1, \ldots, d_k\}$ be a state in $\mathcal{Q}$. The set of terms accepted by $\{d_1, \ldots, d_k\}$ in the determinized DFTA is exactly those terms that are accepted by all of $d_1, \ldots, d_k$ in the original FTA and by *no other state*. This is summarised formally as follows.

Property 1. $\{d_1, \ldots, d_k\} \in \mathcal{Q}$ iff $(L(d_1) \cap \ldots \cap L(d_k)) \setminus \bigcup\{L(d') \mid d' \in Q \setminus \{d_1, \ldots, d_k\}\}$ is nonempty.

Define $\mathsf{dettypes}(d, \mathcal{Q}) = \{d' \mid d' \in \mathcal{Q}, d \in d'\}$. Let $d \in Q$, and let $L_Q(d)$ represent the terms accepted by d in the original FTA. Let $\{d_1, \ldots, d_k\} \in \mathcal{Q}$ and let $L_{\mathcal{Q}}(\{d_1, \ldots, d_k\})$ be the set of terms accepted by $\{d_1, \ldots, d_k\}$ in the corresponding DFTA. Then we have $L_Q(d) = \cup\{L_{\mathcal{Q}}(d') \mid d' \in \mathsf{dettypes}(d, \mathcal{Q})\}$.

Intuitively, $\mathsf{dettypes}(d, \mathcal{Q})$ tells us the set of states in $\mathcal{Q}$ into which d is split during determinization. Thus, the use of sets of states from the original FTA to denote states in the DFTA gives us a convenient way of relating each state in the original FTA with the "equivalent" set of states in the corresponding DFTA.

Example 3. Let $Q = \{list, dynamic\}$ with transitions as defined in Example 2. The determinization algorithm yields states $\mathcal{Q} = \{\{list, dynamic\}, \{dynamic\}\}$ corresponding to *list* and *nonlist* respectively. Then $\mathsf{dettypes}(dynamic, \mathcal{Q}) = \{\{list, dynamic\}, \{dynamic\}\}$ and $\mathsf{dettypes}(list, \mathcal{Q}) = \{\{list, dynamic\}\}$. This shows that the type *list* in the original FTA corresponds to $\{list, dynamic\}$ in the DFTA, while the type *dynamic* is split into two disjoint types $\{list, dynamic\}$ (lists) and $\{dynamic\}$ (non-lists) in the DFTA.

Determinization of the Union of Two FTAs. Let $\langle Q_1, \Sigma, \Delta_1\rangle$ and $\langle Q_2, \Sigma, \Delta_2\rangle$ be FTAs based on the same signature and let $\langle Q_1 \cup Q_2, \Sigma, \Delta_1 \cup \Delta_2\rangle$ be the union of the two automata. (Note, we can assume without loss of generality that Q_1 and Q_2 are disjoint.) Determinize all three automata; we will refer to the sets of states of the respective DFTAs as $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_\star$.

3.1 Correspondence Between Pre-interpretations and DFTAs

A *pre-interpretation* J of a signature Σ is defined by a domain Q_J and a mapping I_J which maps each n-ary function symbol $f/n \in \Sigma$ to a function $Q_J^n \to Q_J$, denoted f_J.

It was shown in [3] that a complete DFTA is equivalent to a *pre-interpretation* of Σ. Thus when we speak of a pre-interpretation we could equally well refer to a completed DFTA and vice versa. Each rule $f(d_1, \ldots, d_n) \to d$ in the DFTA corresponds to an equation $f_J(d_1, \ldots, d_n) = d$ in the pre-interpretation J. The set of rules with the same function f/n on the left defines the function f_J onto which f/n is mapped by I_J. The DFTA is complete, hence the function f_J is total.

Example 4. The DFTA in Example 2 can be written as the pre-interpretation J with domain $\{list, nonlist\}$ and functions $[]_J, [.|.]_J, a_J, s_J$ defined by:

$$\{[]_J = list, [list|list]_J = list, [nonlist|list]_J = list, [list|nonlist]_J = nonlist,\\ [nonlist|nonlist]_J = nonlist, a_J = nonlist,\\ s_J(list) = nonlist, s_J(nonlist) = nonlist\}$$

3.2 Concrete and Abstract Domains of Interpretation

We take the concrete semantic domain of a logic program to be the set of its Herbrand interpretations [7] over an extended signature containing constants corresponding to variables, thus allowing information about term instantiation to be captured [11]. Note that models based on this semantics cannot capture certain properties directly, such as definite freeness. However the semantics provides a useful approximation of the computed answers. Let Σ be the signature of the language of the program, consisting of a set of ranked function and predicate symbols. Let Atom_Σ be the set of atoms of form $p(t_1, \ldots, t_n)$ where $p \in \Sigma$ is an n-ary predicate symbol and $t_1, \ldots, t_n \in \mathsf{Term}_\Sigma$. Atom_Σ is often called the Herbrand base of P. A Herbrand interpretation of P is a subset of Atom_Σ, representing the atoms interpreted as true. The lattice of Herbrand interpretations $\mathcal{D} = \langle 2^{\mathsf{Atom}_\Sigma}, \subseteq, \cup, \cap, \emptyset, \mathsf{Atom}_\Sigma \rangle$ is called the *concrete domain.*

From a Pre-Interpretation to an Abstract Domain. Let J be a pre-interpretation of Σ with domain Q_J. The set Atom_J is the set of expressions $p(d_1, \ldots, d_n)$ where $p \in \Sigma$ is an n-ary predicate symbol and $d_1, \ldots, d_n \in Q_J$. The lattice of interpretations over J, $\mathcal{A}_J = \langle 2^{\mathsf{Atom}_J}, \subseteq, \cup, \cap, \emptyset, \mathsf{Atom}_J \rangle$ is called the *abstract domain based on pre-interpretation* J. A Galois connection between $\mathcal{A}_J$ and $\mathcal{D}$ is given by the concretisation function $\gamma_J : 2^{\mathsf{Atom}_J} \rightarrow 2^{\mathsf{Atom}_\Sigma}$.

$$\gamma_J(S) = \bigcup \{ \{ p(t_1, \ldots, t_n) \mid t_i \in L(d_i), 1 \leq i \leq n \} \mid p(d_1, \ldots, d_n) \in S \}.$$

The expression $L(d_i)$ above refers to the subset of Term_Σ accepted by d_i (considered as the state of a DFTA), as mentioned above.

Although we are not concerned with the semantic function in the present work, we note that the least model $M_J[P]$ of a program P for a pre-interpretation J can be obtained by a fixpoint computation. $M_J[P]$ is an abstraction of the least Herbrand model $M[P]$, in the sense that $\gamma(M_J[P]) \supseteq M[P]$. So any FTA forms the basis for an abstraction of a program in which the meaning of a predicate p is abstracted by a set of domain atoms over the corresponding disjoint types.

Property 2. Let $\mathcal{A}_J$ be the abstract domain constructed from pre-interpretation J. Then each element of the abstract domain represents a unique element of the concrete domain; that is, for all $S_1, S_2 \in 2^{\mathsf{Atom}_J}$, $S_1 = S_2$ iff $\gamma_J(S_1) = \gamma_J(S_2)$. This holds because for all $d_1, d_2 \in Q_J$, $d_1 = d_2$ iff $L(d_1) \cap L(d_2) \neq \emptyset$.

Constructing a Product Domain. Suppose J_1 and J_2 are DFTAs obtained from FTAs $\langle Q_1, \Sigma, \Delta_1 \rangle$ and $\langle Q_2, \Sigma, \Delta_2 \rangle$ respectively.

Let $\mathcal{A}_1 = \langle 2^{\mathsf{Atom}_{J_1}}, \subseteq, \cup, \cap, \emptyset, \mathsf{Atom}_{J_1}\rangle$ and $\mathcal{A}_2 = \langle 2^{\mathsf{Atom}_{J_2}}, \subseteq, \cup, \cap, \emptyset, \mathsf{Atom}_{J_2}\rangle$ be the resulting abstract domains. We form a product domain $\mathcal{A}_\star = \langle 2^{\mathsf{Atom}_{J_\star}}, \subseteq, \cup, \cap, \emptyset, \mathsf{Atom}_{J_\star}\rangle$ where $J_\star$ is the DFTA of the union $\langle Q_1 \cup Q_2, \Sigma, \Delta_1 \cup \Delta_2\rangle$.

Claim. We claim that $\mathcal{A}_\star$ is the reduced cartesian product $\mathcal{A}_1 \star \mathcal{A}_2$ [1].

This claim is informally justified as follows. The reduced product of $\mathcal{A}_1$ and $\mathcal{A}_2$ is defined as the result of applying a reduction operator ρ [1] to the cartesian product of domains of $\mathcal{A}_1$ and $\mathcal{A}_2$ (with elements ordered componentwise). The effect of ρ, informally speaking, is to "bring to the abstract the conjunction of properties we would have in the concrete". Let S_1 and S_2 be elements of $2^{\mathsf{Atom}_{J_1}}$ and $2^{\mathsf{Atom}_{J_2}}$ respectively. Then $\mathcal{A}_\star$ contains a unique element $S_\star$ such that $\gamma_1(S_1) \cap \gamma_2(S_2) = \gamma_\star(S_\star)$. Such an element $S_\star$ exists since the intersection of any two regular types in the original DFTAs is represented in the DFTA of the union. Thus $\mathcal{A}_\star$ is at least as precise as the cartesian product of $\mathcal{A}_1$ and $\mathcal{A}_2$. $S_\star$ is unique (for each S_1 and S_2) due to Property 2. This implies that the reduction operator [1] is the identity function when applied to the product domain.

Lemma 1. *Let J_1, J_2 and $J_\star$ be pre-interpretations constructed as above. Q_1 is the set of states in the FTA used to derive J_1. Let $S_\star \in 2^{\mathsf{Atom}_{J_\star}}$. Then the element $S_1 \in 2^{\mathsf{Atom}_{J_1}}$ defined as $S_1 = \{p(\bar{d}_1 \cap Q_1, \ldots, \bar{d}_n \cap Q_1) \mid p(\bar{d}_1, \ldots, \bar{d}_n) \in S_\star\}$ is the best approximation of $S_\star$ in the domain $2^{\mathsf{Atom}_{J_1}}$. That is, $S_1 = \cap\{S'_1 \mid \gamma_\star(S_\star) \subseteq \gamma_1(S'_1)\}$. (Similarly for $2^{\mathsf{Atom}_{J_2}}$ by symmetry.)*

Proof. The notation $\bar{d}$ means that $\bar{d}$ is a state in a DFTA, i.e. it is a set of states from the original FTA. First show that $\{p(\bar{d}_1 \cap Q_1, \ldots, \bar{d}_n \cap Q_1) \mid p(\bar{d}_1, \ldots, \bar{d}_n) \in S_\star\} \subseteq \cap\{S'_1 \mid \gamma_\star(S_\star) \subseteq \gamma_1(S'_1)\}$. Let $p(\bar{e}_1, \ldots, \bar{e}_n) \in \{p(\bar{d}_1 \cap Q_1, \ldots, \bar{d}_n \cap Q_1) \mid p(\bar{d}_1, \ldots, \bar{d}_n) \in S_\star\}$. Then there exists $p(\bar{d}_1, \ldots, \bar{d}_n) \in S_\star\}$ such that for $1 \leq i \leq n$, $\bar{d}_i = \bar{e}_i \cup \bar{f}$ where $\bar{f} \cap Q_1 = \emptyset$. Then $\gamma_\star(S_\star)$ contains an element $p(t_1, \ldots, t_n)$ such that for $1 \leq i \leq n$, $t_i \in L(\bar{e}_i \cup \bar{f})$, where $\bar{f} \cap Q_1 = \emptyset$. For any element $S'_1 \in 2^{\mathsf{Atom}_{J_1}}$, if $p(t_1, \ldots, t_n) \in \gamma_1(S'_1)$ then $p(\bar{e}_1, \ldots, \bar{e}_n) \in S'_1$ since each $t_i \in L(\bar{e})$ for exactly one $\bar{e} \subseteq Q_1$, and that $\bar{e}$ must be $\bar{e}_i$. Hence $p(\bar{e}_1, \ldots, \bar{e}_n) \in \cap\{S'_1 \mid \gamma_\star(S_\star) \subseteq \gamma_1(S'_1)\}$.

Now show that $\cap\{S'_1 \mid \gamma_\star(S_\star) \subseteq \gamma_1(S'_1)\} \subseteq \{p(\bar{d}_1 \cap Q_1, \ldots, \bar{d}_n \cap Q_1) \mid p(\bar{d}_1, \ldots, \bar{d}_n) \in S_\star\}$. Let $p(\bar{e}_1, \ldots, \bar{e}_n) \in \cap\{S'_1 \mid \gamma_\star(S_\star) \subseteq \gamma_1(S'_1)\}$. If for all $S'_1 \in 2^{\mathsf{Atom}_{J_1}}$ such that $\gamma_\star(S_\star) \subseteq \gamma_1(S'_1)$, $p(\bar{e}_1, \ldots, \bar{e}_n) \in S'_1$, then $S_\star$ contains at least one element $p(\bar{d}_1, \ldots, \bar{d}_n)$ such that for $1 \leq i \leq n$, $\bar{d}_i = \bar{e}_i \cup \bar{f}$ and $\bar{f} \cap Q_1 = \emptyset$. This is because $\gamma_1(\bar{e}_i) \cap \gamma_\star(\bar{d}) = \emptyset$ for all $\bar{d}$ not of the form $\bar{d} = \bar{e} \cup \bar{f}$ and $\bar{f} \cap Q_1 = \emptyset$. Hence $p(\bar{e}_1, \ldots, \bar{e}_n) \in \{p(\bar{d}_1 \cap Q_1, \ldots, \bar{d}_n \cap Q_1) \mid p(\bar{d}_1, \ldots, \bar{d}_n) \in S_\star\}$.

Property 3. A product domain $\mathcal{A}_1 \star \mathcal{A}_2$ can precisely represent elements of its factors $\mathcal{A}_1$ and $\mathcal{A}_2$. That is, if S_1 is an element of the domain of $\mathcal{A}_1$ then there exists an element $S_\star$ in the domain of $\mathcal{A}_\star$ such that $\gamma_1(S_1) = \gamma_\star(S_\star)$. The element is unique by Property 2. Furthermore, $S_1 = \{p(\bar{d}_1 \cap Q_1, \ldots, \bar{d}_n \cap Q_1) \mid p(\bar{d}_1, \ldots, \bar{d}_n) \in S_\star\}$ by Lemma 1.

Example 5. Let $\langle Q_1, \Sigma, \Delta_1\rangle$ and $\langle Q_2, \Sigma, \Delta_2\rangle$ be FTAs, where $Q_1 = \{g, nong\}$ and $Q_2 = \{list, dynamic, int\}$, where these elements have their expected meanings. The DFTA states $\mathcal{Q}_1$ obtained from Q_1 are $\{\{g\}, \{nong\}\}$ and the DFTA states $\mathcal{Q}_2$ obtained from Q_2 are $\{\{dynamic, list\}, \{dynamic, int\}, \{dynamic\}\}$. The states $\mathcal{Q}_\star$ of the DFTA obtained from the union $\langle Q_1 \cup Q_2, \Sigma, \Delta_1 \cup \Delta_2\rangle$ are

$$\{\{dynamic, list, g\}, \{dynamic, list, nong\},\\ \{dynamic, int, g\}, \{dynamic, g\}, \{dynamic, nong\}\}.$$

(Note that there are fewer states in $\mathcal{Q}_\star$ than in the cartesian product of $\mathcal{Q}_1$ and $\mathcal{Q}_2$.) Consider $S_1 = \{p(\{g\}, \{g\}\} \in 2^{\mathsf{Atom}_{J_1}}$. Then the element $S_\star \in 2^{\mathsf{Atom}_{J_\star}}$ such that $\gamma_1(S_1) = \gamma_\star(S_\star)$ is

$$\{p(\{dynamic, list, g\}, \{dynamic, g\}), p(\{dynamic, list, g\}, \{dynamic, int, g\}),\\ p(\{dynamic, list, g\}, \{dynamic, list, g\}), p(\{dynamic, int, g\}, \{dynamic, g\}),\\ p(\{dynamic, int, g\}, \{dynamic, list, g\}), p(\{dynamic, g\}, \{dynamic, list, g\}),\\ p(\{dynamic, int, g\}, \{dynamic, int, g\}), p(\{dynamic, g\}, \{dynamic, int, g\}),\\ p(\{dynamic, g\}, \{dynamic, g\})\}.$$

It contains every possible combination of arguments from the product DFTA states that intersect with $\{g\}$.

In this case, any non-empty subset $S'_\star$ of $S_\star$ also satisfies $S_1 = \{p(\bar{d}_1 \cap Q_1, \ldots, \bar{d}_n \cap Q_1) \mid p(\bar{d}_1, \ldots, \bar{d}_n) \in S'_\star\}$.

3.3 Transformation from One Type Domain to Another

Now we can summarise the general method for representing an element of a domain based on regular types in another domain based on different types.

Proposition 1. *Suppose J_1 and J_2 are two DFTAs (i.e. pre-interpretations) obtained from FTAs $\langle Q_1, \Sigma, \Delta_1\rangle$ and $\langle Q_2, \Sigma, \Delta_2\rangle$ respectively. Let $J_\star$ be the DFTA of the union $\langle Q_1 \cup Q_2, \Sigma, \Delta_1 \cup \Delta_2\rangle$.*

Given $S_1 \in 2^{\mathsf{Atom}_{J_1}}$, let $S_\star \in 2^{\mathsf{Atom}_{J_\star}}$ satisfy $\gamma_1(S_1) = \gamma_\star(S_\star)$. Property 3 shows that this exists (though we did not yet show an explicit procedure for computing it). Then let $S_2 = \{p(\bar{d}_1 \cap Q_2, \ldots, \bar{d}_n \cap Q_2) \mid p(\bar{d}_1, \ldots, \bar{d}_n) \in S_\star\}$. Then $S_2 = \alpha_2(\gamma_1(S_1))$, that is, S_2 is the best representative of S_1 in $2^{\mathsf{Atom}_{J_2}}$.

Proof. $S_2 = \cap\{S'_1 \mid \gamma_\star(S_\star) \subseteq \gamma_2(S'_1)\}$ by Lemma 1. But since we know that $\gamma_1(S_1) = \gamma_\star(S_\star)$ we can write $S_2 = \cap\{S'_1 \mid \gamma_1(S_1) \subseteq \gamma_2(S'_1)\}$. This is equivalent to $S_2 = \alpha_2(\gamma_1(S_1))$ (see Section 2).

4 Construction of a Standard Abstract Domain from the `CiaoPP` Assertion Language

As already mentioned, a set of assertions in the `CiaoPP` assertion language about success of a predicate p/n can be interpreted as a set of abstract atoms of form

$p(d_1, \ldots, d_n)$ where $d_1, \ldots, d_n$ are the names of regular types defined within the `CiaoPP` system. Such a description is called an *abstract success set*. Note that the `CiaoPP` system itself does not present all such values d_i as regular types. There are modes and primitive types as well; however, as shown below, in the context of a particular program and signature we are able to interpret all of these concepts as regular types defined by a finite set of rules. A discussion of the use of regular types to represent modes is contained in previous work [3, 8].

Having expressed all the abstract values d_i as regular types, we will apply the determinization algorithm on the regular types to obtain an abstract domain, as summarised in Section 3. This will be called the *standard domain* for a program.

Example 6. The abstract success sets for some built-in predicates, as contained in the standard assertion database of `CiaoPP`, is shown in the following table.

$atom\_concat/2$	$\{atom\_concat(atm, atm, atm)\}$
$write/2$	$\{write(stream, term)\}$
$length/2$	$\{length(list, int)\}$
$is/2$	$\{is(num, arithexpression)\}$

Here, *atm*, *int* and *num* are primitive types; *stream*, *list* and *arithexpr* are defined by means of regular type rules, and *term* denotes the set of all terms. The rules defining *stream*, for example, are as follows:

$$user\_input \rightarrow stream \qquad user\_output \rightarrow stream$$
$$user\_error \rightarrow stream \qquad user \rightarrow stream$$
$$'\$stream'(int, int) \rightarrow stream$$

We will now show how all such descriptions, including primitive types and those such as *term* can be defined as regular types, in the context of a given program.

4.1 Construction of the Standard FTA

Given a program P, we construct the standard abstract domain. It is based on a finite tree automaton $\langle Q_{std}, \Sigma_{std}, \Delta_{std} \rangle$. We now show how each of these components is made up.

The standard types Q_{std}. The states Q_{std} consist of the following components: (1) a set of *defined system types* Q_s, defined by rule Δ_s; (2) a set of *primitive types* Q_{prim}; (3) a set of *contextual types* Q_{cntxt}.

The defined system types Q_s comprise types that are defined by regular type rules in the `CiaoPP` assertion language. Example include *arithexpr* and *list* as shown in Example 6, which are used in many predicates, and also *keylist*, *lock_mode*, *io_mode*, *stream* and *stream_alias* which concern only one or two library predicates. The primitive types Q_{prim} include *num*, *int*, *flt*, *nnegint* and *atm* which implicitly are defined as infinite (or very large) sets of constants, but are in practice defined by means of a characteristic predicate that is true or false for each constant. The descriptors *gnd*, *nonvar*, *var*, *term* and *struct* are called

contextual types since their definition depends on the particular signature. In CiaoPP these are also handled by means of a characteristic predicate, but unlike primitive types they may be true for non-constants. We define the set of standard types $Q_{std} = Q_s \cup Q_{prim} \cup Q_{cntxt}$.

The global signature of a program Σ_{std}. We now construct Σ_{std}, the global signature, which consists of the following components: (1) the *program signature* Σ_P; (2) the *system signature* Σ_s; (3) the *primitive signature* Σ_{prim}. Note that, unlike Q_{std}, the global signature is dependent on the program to be analysed as well as the standard system functions and constants.

Given a program P to be analysed let Σ_P be the set of function symbols occurring in P and let Σ_s be the set of function symbols occurring in Δ_s, the rules defining Q_s. The *primitive signature* Σ_{prim} is a set of constant symbols that contains sufficient constants to distinguish each of the primitive types Q_{prim}. More precisely, for each non-empty subset $D = \{d_1, \ldots, d_k\}$ of Q_{prim}, let Σ_D be the set of constants that are of type d_i for all $d_i \in D$, and are not of any other type. Then Σ_{prim} contains at least one constant from each non-empty set Σ_D. We also insist that Σ_{prim} is disjoint from $\Sigma_P \cup \Sigma_s$. A typical set of constants in Σ_{prim} is $\{0, 1, 1.0, -1, '\$CONST'\}$. Thus for instance, we know that the set $nnegint \subset int$; therefore Σ_{prim} should contain at least one constant that is in both int and $nnegint$ (e.g. 1) and one which is in int but not in $nnegint$ (e.g. -1). However, note that if P happens to contain the constants 1 or -1 we must pick another member of $int \cap nnegint$ instead of 1 or another member of $int \setminus nnegint$ to replace -1.

We define the global signature $\Sigma_{std} = \Sigma_P \cup \Sigma_s \cup \Sigma_{prim} \cup \{'\$VAR'\}$ where $'\$VAR'$ is a constant that does not appear in any other component of Σ_{std}. We will discuss the role of $'\$VAR'$ when constructing the contextual type rules.

The global type rules Δ_{std}. The set of type rules defining the types Q_{std} over the signature Σ_{std} consists of the following components: (1) the *system type rules* Δ_s; (2) the *primitive rules* Δ_{prim}; (3) the *contextual type rules* Δ_{cntxt}.

The system type rules Δ_s are simply extracted from the CiaoPP system. For the primitive rules we assume that the Prolog system provides some built-in predicate for testing whether a given constant is of a given primitive type. Hence given the signature Σ_{std} we can enumerate the set of rules Δ_{prim} of the form $c \rightarrow d$ where $c \in \Sigma_{std} \setminus \{'\$VAR'\}$ is a constant, $d \in Q_{prim}$ and c is of type d.

The types in Q_{cntxt} are those whose definitions depend on the signature, such as *gnd*. Given the global signature Σ_{std}, then Δ_{cntxt} is a set of rules defining each type in Q_{cntxt} in terms of Σ_{std}. The details of the rules for *gnd*, *nonvar*, *var*, *term* and *struct* are as follows.

- $f(gnd, \ldots, gnd) \rightarrow gnd$, for each n-ary function $f \in \Sigma_{std} \setminus \{'\$VAR'\}$;
- $f(term, \ldots, term) \rightarrow nonvar$, for each n-ary function $f \in \Sigma_{std} \setminus \{'\$VAR'\}$;
- $f(term, \ldots, term) \rightarrow term$, for each n-ary function $f \in \Sigma_{std}$;
- $f(term, \ldots, term) \rightarrow struct$, for each n-ary function $(n > 0)$ $f \in \Sigma_{std} \setminus \{'\$VAR'\}$;
- the single rule $'\$VAR' \rightarrow var$.

Note the role of $'\$VAR'$; it is a constant that appears in the type *term* and *var* but no other type. The idea is to distinguish the general type *term* from other types (such as *gnd*), by including a constant $'\$VAR'$ that no other type contains, apart from *var*, which only contains $'\$VAR'$. Thus a predicate argument that is specified as *term* can contain any term (since $L(term) \supset L(d)$ for all d) including terms that are of no other type. This technique has been used to model the presence of variables in previous work applying pre-interpretations for program analysis [9–11].

Determinization of the Standard FTA. Having constructed the finite tree automaton $\langle Q_{std}, \Sigma_{std}, \Delta_{std}\rangle$ we can build a pre-interpretation and hence an abstract domain, called the *standard domain*, as described in Section 3. In the DFTA obtained by determinizing $\langle Q_{std}, \Sigma_{std}, \Delta_{std}\rangle$ the states are elements of $2^{Q_{std}}$. In the worst case, the set of states would be $2^{|Q_{std}|} - 1$, but it turns out to be much less. The number of DFTA states is in fact 37, almost the same as the size of Q_{std}. We can produce a compact representation for Δ_{std}. The number of transitions in the DFTA, if represented explicitly, would be very large (24,239) but we use a compact representation as discussed in [12]. The determinization procedure takes approximately 0.6 seconds. In fact the conversion procedure does not use the DFTA rules, but relies only on the set of states of the DFTA, thanks to the use of the representation of states as elements of $2^{Q_{std}}$.

Representation of Abstract Success Sets in the Standard Domain. An abstract success set obtained from the `CiaoPP` assertion database, such as those shown in Example 6, can be represented as an element of the standard domain. Let M^p be the abstract success set of some predicate p. Let $\mathcal{Q}_{std}$ be the set of states in the standard DFTA. Then the representation of M^p in the standard domain is

$$M^p_{std} = \{p(d'_1, \ldots, d'_n) \mid \begin{array}{l} p(d_1, \ldots, d_n) \in M^p \\ \quad \wedge\, \forall i : 1 \leq i \leq n : d'_i \in \mathsf{dettypes}(d_i, \mathcal{Q}_{std})\}. \end{array}$$

M^p_{std} is an exact representation of M^p as formalised by the following property.

Property 4. Let γ be the concretisation function in the standard domain. Then $\gamma(M^p_{std}) = \{p(t_1, \ldots, t_n) \mid p(d_1, \ldots, d_n) \in M^p, t_i \in L(d_i), 0 \leq i \leq n\}$. Here $L(d_i)$ refers to the set of terms accepted by d_i in the standard FTA. The property follows from the definition of $\mathsf{dettypes}$.

5 The User Domain and the Construction of the Product Domain

Now we turn to the question of converting the descriptions of predicates given with respect to the states of the standard FTA into descriptions in terms of some other, user-supplied FTA. Let $\langle Q_u, \Sigma_u, \Delta_u\rangle$ be an FTA given by the user. We assume that $\Sigma_u \subseteq \Sigma_{std}$. (This is no loss of generality since the program P can

always to modified to contain more function symbols without affecting its intended behaviour, e.g. by adding a dummy clause containing the required function symbols.) Therefore we consider the FTA with the full signature $\langle Q_u, \Sigma_{std}, \Delta'_u \rangle$. Q_u also contains the contextually defined type *dynamic* and possibly other contextual types; Δ'_u is obtained from Δ_u by extending the definitions of those contextual types for the full signature Σ_{std}. We also assume that if a type appears both in Q_u and Q_{std} then it has the same meaning in both.

The intention is to analyse the given program P with respect to the pre-interpretation obtained by determinizing the user-supplied FTA. The *user domain* for analysis is based on the DFTA obtained from $\langle Q_u, \Sigma_{std}, \Delta'_u \rangle$ as described in Section 3.

We construct an FTA combining the standard FTA with the user-supplied FTA $\langle Q_u, \Sigma_{std}, \Delta'_u \rangle$. The *union FTA* $\langle Q_{std} \cup Q_u, \Sigma_{std}, \Delta'_u \cup \Delta_{std} \rangle$ is determinized, and the *product abstract domain* is the abstract domain obtained from the resulting DFTA.

5.1 Converting Abstract Success Sets to the User Domain

As already discussed, we are supplied with an abstract success set of each of the external predicates p/n in P, as a set of atoms of form $p(d_1, \ldots, d_n)$ where $d_1, \ldots, d_n \in Q_{std}$. (We can safely assume a default abstraction where all arguments are of type *term*, if no abstraction is defined).

Representing an abstract success set in the product domain. Let p/n be a predicate and let its abstraction over Q_{std} be M^p. Let $\mathcal{Q}$ be the set of states in the product DFTA that is obtained from the standard FTA and some user FTA. Then the corresponding abstract success set, defined over the set of determinized types $\mathcal{Q}_\star$, is defined as

$$M^p_\star = \{p(d'_1, \ldots, d'_n) \mid \begin{array}{l} p(d_1, \ldots, d_n) \in M^p \\ \quad \wedge\, \forall i : 1 \leq i \leq n : d'_i \in \mathsf{dettypes}(d_i, \mathcal{Q}_\star)\}. \end{array}$$

Property 5. Let γ be the concretisation function in the product domain. Then $\gamma(M^p_\star) = \{p(t_1, \ldots, t_n) \mid p(d_1, \ldots, d_n) \in M^p, t_i \in L(d_i), 0 \leq i \leq n\}$. Here $L(d_i)$ refers to the set of terms accepted by d_i in the standard FTA. As for Property 4 this shows that an abstract model has an exact representation in the product domain.

Example 7. Let the given abstract success set of $length/2$ be $\{length(list, int)\}$. Let the user FTA be the following definitions of the types *matrix* and *row* along with the rules $f(dynamic, \ldots, dynamic) \rightarrow dynamic$ for each $f/n \in \Sigma_{std}$.

$$\begin{array}{ll} [] \rightarrow row & [] \rightarrow matrix \\ [dynamic|row] \rightarrow row & [row|matrix] \rightarrow matrix \end{array}$$

Then the abstract success set in the product domain includes 32 abstract atoms. An example of an atom in the set is

$$length(\{callable, list, struct, term, dynamic, row, sourcename, struct, term\},\\ \{arithexpression, callable, character\_code, constant, gnd, int, nnegint,\\ num, struct, term, dynamic, sourcename, atom\_or\_number\})$$

The first argument represents one of the disjoint types that make up the type *list* in the product DFTA, and similarly the second argument is a part of the *int* type.

Projecting a model onto user types. The final stage is to project the model from the product domain onto the user domain. Let $p \in E_P$ be an external predicate and let $M^p_\star$ be the model of p over the determinized types T'. Then the projection of $M^p_\star$ onto the user types Q_u is defined as $M^p_u = \{p(d'_1 \cap Q_u, \ldots, d'_n \cap Q_u) \mid p(d'_1, \ldots, d'_n) \in M^p_\star\}$, Note that we can ensure that each argument $d'_1 \cap Q_u$ is non-empty by including *dynamic* in Q_u.

Example 8. Let $M^{length}_\star$ be the model of *length* in the product domain as in the previous example. Let $Q_u = \{matrix, row, dynamic\}$. Then the projection of $M^{length}_\star$ onto Q_u is

$$\{length(\{row, dynamic\}, \{dynamic\}),\\ length(\{matrix, row, dynamic\}, \{dynamic\})\}$$

The projected model is not expressed directly in the set of user types Q_u but rather in the disjoint types resulting from determinizing Q_u. The model expressed in this form is exactly what is required for computing a model of the program P over the user types, using the approach in [3].

The projected models are safe approximations of the models over the standard types, and are the best available approximations in the user domain.

Proposition 2. *Let P be a program and let p be an externally defined predicate occurring in P. Let M^p be an abstraction of the success set of p over the standard types Q_{std} and let M^p_{std} be the exact representation of M^p in the standard domain. Let M^p_u be the projection onto the user types Q_u. Then $M^p_u = \alpha_u(\gamma_{std}(M^p_{std}))$, which is the best available safe approximation of M^p_{std} in the user domain.*

Proof. This is a direct consequence of Proposition 1 and Property 5.

6 Implementation and Experiments

We have implemented the procedure in Ciao-Prolog and used it to compute built-in tables for a range of built-ins over simple domains, such as the Pos domain and the default domain used with the binding time analysis tool for the Logen system [8].

Note that the results obtained are not always the best possible for a given domain. This is due to two main causes. Firstly, the assertion database of `CiaoPP` is not yet complete. Secondly, even where values have been entered they do not always capture dependencies between arguments. For example, for the list append

Table 1. Standard Abstract Models

Predicate	Standard model
is/2	is(num,arithexpression)
number/1	number(num)
member/2	member(term,struct)
length/2	length(list,int),
=../2	=..(term,list)
write/1	write(stream,term)
atom_concat/3	atom_concat(atm,atm,atm)

predicate $app/3$ the given abstraction might be $\{app(list, list, list)\}$. Since a *list* can be either ground or non-ground, we cannot derive an accurate description of *app* over the Pos domain from the given information. The optimal result would be $\{app(g, g, g), app(g, nong, nong), app(nong, g, nong), app(nong, nong, nong)\}$, but our procedure will return the most general model having all eight possible combinations of g, *nong* arguments.

We show in Table 1 the abstract models of certain predicates extracted from the `CiaoPP` database. Table 2 shows the derived models over the FTA defining the types *dynamic* and *static* (which denote the same as *gnd* and *term* in the standard models, but are the names used in the binding time analysis of Logen). The DFTA has two states $\{\{dynamic, static\}, \{dynamic\}\}$ denoting ground and non-ground terms respectively. An underscore stands for either state. This domain is equivalent to Pos [13] but the models derived in Table 2 are not the best possible within the domain, though they are optimal with respect to the given standard models. Table 3 shows the derived models over the FTA defining the types *dynamic* and *var*. The corresponding DFTA contains states $\{\{dynamic, var\}, \{dynamic\}\}$ denoting variable and non-variable terms respectively.

Regarding performance and scalability, we remark that so far we handle the full set of types from the `CiaoPP` database without problems, using a Prolog implementation. The conversion is performed off-line, not during analysis, so

Table 2. Models over {dynamic,static} (*ground*) and {dynamic} (*non-ground*)

Predicate	Abstract model for types $\{dynamic, static\}$
is/2	is({dynamic,static},{dynamic,static})
number/1	number({dynamic,static})
member/2	member(_,_)
length/2	length({dynamic},{dynamic,static}) length({dynamic,static},{dynamic,static})
=../2	=..(_,_)
write/1	write({dynamic,static},{dynamic}) write({dynamic,static},{dynamic,static})
atom_concat/3	atom_concat({dynamic,static},{dynamic,static},{dynamic,static})

Table 3. Models over {dynamic,var} (*var*) and {dynamic} (*non-var*)

Predicate	Abstract model for types $\{dynamic, var\}$
is/2	is({dynamic},{dynamic})
number/1	number({dynamic})
member/2	member(_,{dynamic}),
length/2	length({dynamic},{dynamic})
=../2	=..(_,{dynamic})
write/1	write({dynamic},{dynamic,var}) write({dynamic},{dynamic})
atom_concat/3	atom_concat({dynamic},{dynamic},{dynamic})

absolute time is not critical. However, so far the target user domains have been small. The efficient determinization algorithm described in [12] performs well and we do not anticipate problems moving to larger domains. Scalability issues do arise in representing the models themselves, in domains based on DFTAs with a large number of states. Compact representations of relations using techniques such as BDDs [14, 15] seem to be promising approaches to this problem, and we have already made use of BDDs in handling DFTAs [12].

7 Conclusions and Future Work

We have described a method for translating abstract descriptions of success sets of predicates from a general purpose assertion language, the `CiaoPP` assertion language, into any regular type-based abstract domain. Current work is directed towards automatic translation of the assertions for all standard library predicates into commonly used domains. Effort also needs to be put into completing and making more precise the existing assertions on predicates. We have described the method applied to success set descriptions, but the method applies to call patterns, or backwards analyses, provided that the abstract domain is based on regular types.

We believe that this work also underlines the generality and versatility of regular types for constructing analysis domains. The fact that special purpose, program specific domains can be constructed easily makes it all the more relevant to be able to render information about imported code in such domains, as this work does. Future work will continue to explore the potential of regular type domains and combine them with other domains such as numeric domains.

Acknowledgements. We thank Patrick Cousot for some enlightening remarks, and the LOPSTR referees for useful comments on the extended abstract. This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the FP5 IST-2001-38059 *ASAP* and FP6 IST-15905 *MOBIUS* projects and by the Spanish Ministry of Science and Education under the TIC 2002-0055 *CUBICO* project. J. Gallagher's research is supported in part by the IT-University of Copenhagen.

References

1. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio,Texas, ACM Press, New York, U.S.A. (1979) 269–282
2. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2) (2005)
3. Gallagher, J.P., Henriksen, K.S.: Abstract domains based on regular types. In Lifschitz, V., Demoen, B., eds.: Proceedings of the International Conference on Logic Programming (ICLP'2004). LNCS 3132. (2004) 27–42
4. Filè, G., Giacobazzi, R., Ranzato, F.: A unifying view on abstract domain design. ACM Computing Surveys **28**(2) (1996) 333–336
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In Sagiv, S., ed.: ESOP. LNCS 3444. (2005) 21–30
6. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications. http://www.grappa.univ-lille3.fr/tata (1999)
7. Lloyd, J.: Foundations of Logic Programming: 2nd Edition. Springer-Verlag (1987)
8. Craig, S., Gallagher, J.P., Leuschel, M., Henriksen, K.S.: Fully automatic binding time analysis for Prolog. In Etalle, S., ed.: Pre-Proceedings, 14th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR 2004, Verona, August 2004. (2004) 61–70
9. Boulanger, D., Bruynooghe, M., Denecker, M.: Abstracting *s*-semantics using a model-theoretic approach. In Hermenegildo, M., Penjam, J., eds.: Proc. 6^{th} International Symposium on Programming Language Implementation and Logic Programming, PLILP'94. LNCS 844 (1994) 432–446
10. Boulanger, D., Bruynooghe, M.: A systematic construction of abstract domains. In Le Charlier, B., ed.: Proc. First International Static Analysis Symposium, SAS'94. LNCS 864 (1994) 61–77
11. Gallagher, J.P., Boulanger, D., Sağlam, H.: Practical model-based static analysis for definite logic programs. In Lloyd, J.W., ed.: Proc. of International Logic Programming Symposium, MIT Press (1995) 351–365
12. Gallagher, J.P., Henriksen, K.S., Banda, G.: Techniques for scaling up analyses based on pre-interpretations. In Gabbrielli, M., Gupta, G., eds.: Proceedings of the 21st International Conference on Logic Programming, ICLP'2005. LNCS 3668 (2005). 280–296
13. Marriott, K., Søndergaard, H.: Precise and efficient groundness analysis for logic programs. LOPLAS **2(1-4)** (1993) 181–196
14. Iwaihara, M., Inoue, Y.: Bottom-up evaluation of logic programs using binary decision diagrams. In Yu, P.S., Chen, A.L.P., eds.: ICDE, IEEE Computer Society (1995) 467–474
15. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Pugh, W., Chambers, C., eds.: PLDI, ACM (2004) 131–144