Abstract Domains Based on Regular Types

Gallagher, John Patrick; Henriksen, Kim Steen

Published in:
20th International Conference, ICLP 2004, Proceedings

Publication date:
2004

Document Version
Også kaldet Forlagets PDF

Citation for published version (APA):
Gallagher, J. P., & Henriksen, K. S. (2004). Abstract Domains Based on Regular Types. I B. Demoen, & V. Lifschitz (red.), *20th International Conference, ICLP 2004, Proceedings* Springer.

Abstract Domains Based on Regular Types *

John P. Gallagher and Kim S. Henriksen

Computer Science, Building 42.1, P.O. Box 260, Roskilde University, DK-4000 Denmark
Email: {jpg,kimsh}@ruc.dk

Abstract. We show how to transform a set of regular type definitions into a finite pre-interpretation for a logic program. The derived pre-interpretation forms the basis for an abstract interpretation. The core of the transformation is a determinization procedure for non-deterministic finite tree automata. This approach provides a flexible and practical way of building program-specific analysis domains. We argue that the constructed domains are condensing: thus goal-independent analysis over the constructed domains loses no precision compared to goal-dependent analysis. We also show how instantiation modes such as *ground*, *variable* and *non-variable* can be expressed as regular types and hence integrated with other regular types. We highlight applications in binding time analysis for offline partial evaluation and infinite-state model checking. Experimental results and a discussion of complexity are included.

1 Background

There is a well-established connection between regular types and finite tree automata (FTAs) [1], although typical regular type definition notations [2, 3] usually correspond to top-down deterministic FTAs, which are less expressive than FTAs in general. We show how to build an analysis domain from any FTA on a given program's signature, by transforming it to a *pre-interpretation* for the signature. The main contribution of this paper is thus to link the rich descriptive framework of arbitrary FTAs, which includes modes and regular types, to the analysis framework based on pre-interpretations, and demonstrate the practicality of the link and the precision of the resulting analyses.

In Section 2, we introduce the essential concepts from types and FTAs [4] and a review of the approach to logic program analysis based on pre-interpretations [5–7]. In Section 3 it is shown how to determinize a given FTA on order to construct a pre-interpretation. Section 4 contains some examples. Implementation and complexity issues are discussed in Section 5.

An informal example is given first, to give some intuition into the procedure.

Example 1. Given a program containing functions $[\,]$, $[\_|\_]$ and unary function f, let the type $list$ be defined by $list = [\,]; [any|list]$ and the type any as $any =$

* Work supported in part by European Framework 5 Project ASAP (IST-2001-38059), and the IT-University of Copenhagen.

$f(any)$; [ ]; $[any|any]$. Clearly *list* and *any* are not disjoint: *any* includes *list*. We can *determinize* the types, yielding two types *list* and *nonlist* (l and nl) with type rules $l = [\,]; [nl|l]; [l|l]$ and $nl = [nl|nl]; [l|nl]; f(l); f(nl)$. Every ground term in the language of the program is in exactly one of the two types.

Computing a model of the usual *append* program over the elements l and nl (using a procedure to be described) we will obtain the following "abstract model" of the relation *append*/3: $\{append(l, l, l), append(l, nl, nl)\}$. This can be interpreted as showing that for any correct answer to $append(X, Y, Z)$, X is a *list*, and Y and Z are of the same type (*list* or *nonlist*). Note that it is not possible to express the same information using only *list* and *any*, since whenever *any* is associated with an argument position, its subtype *list* is automatically associated with that argument too. Hence a description of the model of the program using the original types could be no more precise than $\{append(list, any, any)\}$.

2 Preliminaries

Let Σ be a set of function symbols. Each function symbol in Σ has a rank (arity) which is a natural number. Whenever we write an expression such as $f(t_1, \ldots, t_n)$, we assume that $f \in \Sigma$ and has arity n. We write f^n to indicate that function symbol f has arity n. If the arity of f is 0 we write the term $f()$ as f and call f a *constant*. The set of *ground terms* (or *trees*) Term_Σ associated with Σ is the least set containing the constants and all terms $f(t_1, \ldots, t_n)$ such that $t_1, \ldots, t_n$ are elements of Term_Σ and $f \in \Sigma$ has arity n.

A *finite tree automaton (FTA)* is a means of finitely specifying a possibly infinite set of terms. An FTA is defined as a quadruple $\langle Q, Q_f, \Sigma, \Delta\rangle$, where Q is a finite set of *states*, $Q_f \subseteq Q$ is the set of accepting (or final) states, Σ is a set of ranked function symbols and Δ is a set of *transitions*. Each element of Δ is of the form $f(q_1, \ldots, q_n) \to q$, where $f \in \Sigma$ and $q, q_1, \ldots, q_n \in Q$.

FTAs can be "run" on terms in Term_Σ; a successful run of a term and an FTA is one in which the term is *accepted* by the FTA. When a term is accepted, it is accepted by one or more of the final states of the FTA. Different runs may result in acceptance by different states. At each step of a successful bottom-up run, some subterm identical to the left-hand-side of some transition is replaced by the right-hand-side, until eventually the whole term is reduced to some accepting state. The details can be found elsewhere [4]. Implicitly, a tree automaton R defines a set of terms (or tree language), denoted $L(R)$, which is the set of all terms that it accepts.

Tree Automata and Types An accepting state of an FTA can be regarded as a type. Given an automaton $R = \langle Q, Q_f, \Sigma, \Delta\rangle$, and $q \in Q_f$, define the automaton R_q to be $\langle Q, \{q\}, \Sigma, \Delta\rangle$. The language $L(R_q)$ is the set of terms corresponding to type q. A term t *is of type* q if and only if $t \in L(R_q)$.

A transition $f(q_1, \ldots, q_n) \to q$, when regarded as a type rule, is usually written the other way around, as $q \to f(q_1, \ldots, q_n)$. Furthermore, all the rules defining the same type, $q \to R_1, \ldots, q \to R_n$ are collected into a single equation

of the form $q = R_1; \ldots; R_n$. When speaking about types we will usually follow the type notation, but when discussing FTAs we will use the notation for transitions, in order to make it easier to relate to the literature.

Example 2. Let $Q = \{listnat, nat\}$, $Q_f = \{listnat\}$, $\Sigma = \{[], [\_|\_], s^1, 0^0\}$, $\Delta = \{[] \to listnat, [nat|listnat] \to listnat, 0 \to nat, s(nat) \to nat\}$. The type $listnat$ is the set of lists of natural numbers in successor notation; the type rule notation is $listnat = []; [nat|listnat]$, and $nat = 0; s(nat)$.

Let $\Sigma = \{[], [\_|\_], s^1, 0^0\}$, $Q = \{zero, one, list_0, list_1\}$, $Q_f = \{list_1\}$, and $\Delta = \{[] \to list_1, [one|list_1] \to list_1, [zero|list_0] \to list_1, [] \to list_0, [zero|list_0] \to list_0, 0 \to zero, s(zero) \to one\}$, or $list_1 = []; [one|list_1]; [zero|list_0]$ and $list_0 = []; [zero|list_0]$. The type $list_1$ is the set of lists consisting of zero or more elements $s(0)$ followed by zero or more elements 0 (such as $[s(0), 0]$, $[s(0), s(0), 0, 0, 0]$, $[0, 0], [s(0)], \ldots$). This kind of set is not normally thought of as a type.

Deterministic and Non-deterministic Tree Automata It can be shown that (so far as expressiveness is concerned) we can limit our attention to FTAs in which the set of transitions Δ contains no two transitions with the same left-hand-side. These are called *bottom-up deterministic* finite tree automata. For every FTA R there exists a bottom-up deterministic FTA R' such that $L(R) = L(R')$. The sets of terms accepted by states of bottom-up deterministic FTAs are disjoint. Each term in $L(R')$ is accepted by exactly one state.

An automaton $R = \langle Q, Q_f, \Sigma, \Delta \rangle$ is called *complete* if for all n-ary functions $f \in \Sigma$ and states $q_1, \ldots, q_n \in Q$, it contains a transition $f(q_1, \ldots, q_n) \to q$. We may always extend an FTA $\langle Q, Q_f, \Sigma, \Delta \rangle$ to make it complete, by adding a new state $q^\flat$ to Q. Then add transitions of the form $f(q_1, \ldots, q_n) \to q^\flat$ for every combination of f and states $q_1, \ldots, q_n$ (including $q^\flat$) that does not appear in Δ. A complete bottom-up deterministic finite tree automaton in which every state is an accepting state partitions the set of terms into disjoint subsets (types), one for each state. In such an automaton $q^\flat$ can be thought of as the error type, that is, the set of terms not accepted by any other type.

Example 3. Let $\Sigma = \{[], [\_|\_], 0^0\}$, and let $Q = \{list, listlist, any\}$. We define the set Δ_{any}, for a given Σ, to be the set of transitions $\{f(\overbrace{any, \ldots, any}^{n \text{ times}}) \to any \mid f^n \in \Sigma\}$. Let $Q_f = \{list, listlist\}$, $\Delta = \{[] \to list, [any|list] \to list, [] \to listlist, [list|listlist] \to listlist\} \cup \Delta_{any}$. The type $list$ is the set of lists of any terms, while the type $listlist$ is the set of lists whose elements are of type $list$; note that $list$ includes $listlist$.

The automaton is not bottom-up deterministic; for example, three transitions have the same left-hand-side, namely, $[] \to list$, $[] \to listlist$ and $[] \to any$. So for example the term $[[0]]$ is accepted by $list, listlist$ and any. A determinization algorithm can be applied, yielding the following. q_1 corresponds to the type $any \cap list \cap listlist$, $q2$ to the type $(list \cap any) - listlist$, and q_3 to $any - (list \cup listlist)$. Thus q_1, q_2 and q_3 are disjoint. The automaton is given by $Q = \{q_1, q_2, q_3\}$, Σ as before, $Q_f = \{q_1, q_2\}$ and $\Delta = \{[] \to q_1, [q_1|q_1] \to q_1, [q_2|q_1] \to q_1, [q_1|q_2] \to$

$q_2, [q_2|q_2] \to q_2, [q_3|q_2] \to q_2, [q_3|q_1] \to q_2, [q_2|q_3] \to q_3, [q_1|q_3] \to q_3, [q_3|q_3] \to q_3, 0 \to q_3\}$. This automaton is also complete; the determinization of this example will be discussed in more detail in Section 3.

An FTA is *top-down deterministic* if it has no two transitions with both the same right-hand-side and the same function symbol on the left-hand-side. Top-down determinism introduces a loss in expressiveness. It is *not* the case that for each FTA R there is a top-down deterministic FTA R' such that $L(R) = L(R')$. Note that a top-down deterministic automaton can be transformed to an equivalent bottom-up deterministic automaton, as usual, but the result might not be top-down deterministic.

Example 4. Take the second automaton from Example 2. This is not top-down deterministic, due to the presence of transitions $[one|list_1] \to list_1, [zero|list_0] \to list_1$. No top-down deterministic automaton can be defined that has the same language. Thus the set accepted by $list_1$ could not be defined as a type, using type notations that require top-down deterministic rules (e.g. [2, 3]).

Example 5. We define the set Δ_{any} as before. Consider the automaton with transitions $\Delta_{any} \cup \{[] \to list, [any|list] \to list\}$. This is top-down deterministic, but not bottom-up deterministic (since $[] \to list$ and $[] \to any$ both occur). Determinizing this automaton would result in one that is not top-down deterministic.

2.1 Analysis Based on Pre-Interpretations

We now define the analysis framework for logic programs. Bottom-up declarative semantics captures the set of logical consequences (or a model) of a program. The standard, or concrete semantics is based on the Herbrand pre-interpretation. The theoretical basis of this approach to static analysis of definite logic programs was set out in [5, 6] and [7]. We follow standard notation for logic programs [8].

Let P be a definite program and Σ the signature of its underlying language L. A *pre-interpretation* of L consists of

1. a non-empty domain of interpretation D;
2. an assignment of an n-ary function $D^n \to D$ to each n-ary function symbol in Σ ($n \geq 0$).

Correspondence of FTAs and Pre-Interpretations A pre-interpretation with a finite domain D over a signature Σ is equivalent to a complete bottom-up deterministic FTA over the same signature, as follows.

1. The domain D is the set of states of the FTA.
2. Let $\hat{f}$ be the function $D^n \to D$ assigned to $f \in \Sigma$ by the pre-interpretation. In the corresponding FTA there is a set of transitions $f(d_1, \ldots, d_n) \to d$, for each $d_1, \ldots, d_n, d$ such that $\hat{f}(d_1, \ldots, d_n) = d$. Conversely the transitions of a complete bottom-up deterministic FTA define a function [4].

Semantics parameterized by a pre-interpretation We quote some definitions from Chapter 1 of [8]. Let J be a pre-interpretation of L with domain D. Let V be a mapping assigning each variable in L to an element of D. A *term assignment* $T_J^V(t)$ is defined for each term t as follows:

1. $T_J^V(x) = V(x)$ for each variable x.
2. $T_J^V(f(t_1, \ldots, t_n)) = f'(T_J^V(t_1), \ldots, T_J^V(t_n)),\ (n \geq 0)$ for each non-variable term
   $f(t_1, \ldots, t_n)$, where f' is the function assigned by J to f.

Let J be a pre-interpretation of a language L, with domain D, and let p be an n-ary function symbol from L. Then a *domain atom* for J is any atom $p(d_1, \ldots, d_n)$ where $d_i \in D$, $1 \leq i \leq n$. Let $p(t_1, \ldots, t_n)$ be an atom. Then a *domain instance* of $p(t_1, \ldots, t_n)$ with respect to J and V is a domain atom $p(T_J^V(t_1), \ldots, T_J^V(t_n))$. Denote by $[A]_J$ the set of all domain instances of A with respect to J and some V.

The definition of domain instance extends naturally to formulas. In particular, let C be a clause. Denote by $[C]_J$ the set of all domain instances of the clause with respect to J.

Core bottom-up semantics function T_P^J The core bottom-up declarative semantics is parameterised by a pre-interpretation of the language of the program. Let P be a definite program, and J a pre-interpretation of the language of P. Let $Atom_J$ be the set of domain atoms with respect to J. The function $T_P^J : 2^{Atom_J} \to 2^{Atom_J}$ is defined as follows.

$$T_P^J(I) = \left\{ A' \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ A' \leftarrow B'_1, \ldots, B'_n \in [A \leftarrow B_1, \ldots, B_n]_J \\ \{B'_1, \ldots, B'_n\} \subseteq I \end{array} \right\}$$

$\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$: $\mathsf{M}^J[\![P]\!]$ is the minimal model of P with pre-interpretation J.

Concrete Semantics The usual semantics is obtained by taking J to be the Herbrand pre-interpretation, which we call H. Thus $Atom_H$ is the Herbrand base of (the language of) P and $\mathsf{M}^H[\![P]\!]$ is the minimal Herbrand model of P.

The minimal Herbrand model consists of ground atoms. In order to capture information about the occurrence of variables, we extend the signature with an infinite set of extra constants $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$. The Herbrand pre-interpretation over the extended language is called HV. The model $\mathsf{M}^{HV}[\![P]\!]$ is our concrete semantics.

The elements of $\mathcal{V}$ do not occur in the program or goals, but can appear in atoms in the minimal model $\mathsf{M}^{HV}[\![P]\!]$. Let $\mathcal{C}(P)$ be the set of all atomic logical consequences of the program P, known as the Clark semantics [9]; that is, $\mathcal{C} = \{A \mid P \models \forall A\}$, where A is an atom. Then $\mathsf{M}^{HV}[\![P]\!]$ is isomorphic to $\mathcal{C}(P)$. More precisely, let Ω be some fixed bijective mapping from $\mathcal{V}$ to the variables

in L. Let A be an atom; denote by $\Omega(A)$ the result of replacing any constant v_j in A by $\Omega(v_j)$. Then $A \in \mathsf{M}^{HV}[\![P]\!]$ iff $P \models \forall(\Omega(A))$. By taking the Clark semantics as our concrete semantics, we can construct abstractions capturing the occurrence of variables. This version of the concrete semantics is essentially the same as the one discussed in [7].

In our applications, we will always use pre-interpretations that map all elements of $\mathcal{V}$ onto the same domain element, say d_v. In effect, we do not distinguish between different variables. Thus, a pre-interpretation includes an infinite mapping $\{v_0 \mapsto d_v, v_1 \mapsto d_v, \ldots\}$. For such interpretations, we can take a simpler concrete semantics, in which the set of extra constants $\mathcal{V}$ contains just one constant v instead of an infinite set of constants. Then pre-interpretations are defined which include a single mapping $\{v \mapsto d_v\}$ to interpret the extra constant. Examples are shown in Section 4.

Abstract Interpretations Let P be a program and J be a pre-interpretation. Let $Atom_J$ be the set of domain atoms with respect to J. The *concretisation function* $\gamma : 2^{Atom_J} \rightarrow 2^{Atom_{HV}}$ is defined as $\gamma(S) = \{\ A \mid [A]_J \subseteq S\ \}$

$\mathsf{M}^J[\![P]\!]$ is an abstraction of the atomic logical consequences of P, in the following sense.

Proposition 1. *Let P be a program with signature Σ, and $\mathcal{V}$ be a set of constants not in Σ (where $\mathcal{V}$ can be either infinite or finite). Let HV be the Herbrand interpretation over $\Sigma \cup \mathcal{V}$ and J be any pre-interpretation of $\Sigma \cup \mathcal{V}$. Then $\mathsf{M}^{HV}[\![P]\!] \subseteq \gamma(\mathsf{M}^J[\![P]\!])$.*

Thus, by defining pre-interpretations and computing the corresponding least model, we obtain safe approximations of the concrete semantics.

Condensing Domains The property of being a *condensing* domain [10] has to do with precision of goal-dependent and goal-independent analyses (top-down and bottom-up) over that domain. Goal-independent analysis over a condensing domain loses no precision compared with goal-dependent analysis; this has advantages since a single goal-independent analysis can be reused to analyse different goals (relatively efficiently) with the same precision as if the individual goals were analysed.

The abstract domain is 2^{Atom_J}, namely, sets of abstract atoms with respect to the domain of the pre-interpretation J, with set union as the upper bound operator. The conditions satisfied by a condensing domain are usually stated in terms of the abstract unification operation (namely that it should be idempotent and commutative) and the upper bound $\sqcup$ on the domain (which should satisfy the property $\gamma(X \sqcup Y) = \gamma(X) \cup \gamma(Y)$). The latter condition is clearly satisfied ($\sqcup = \cup$) in our domain). Abstract unification is not explicitly present in our framework. However, we argue informally that the declarative equivalent is the abstraction of the equality predicate $X = Y$. This is the set $\{d = d \mid d \in D_J\}$ where D_J is the domain of the pre-interpretation. This satisfies an idempotency property, since for example the clause $p(X, Y) \leftarrow X = Y, X = Y$ gives the same

result as $p(X,Y) \leftarrow X = Y$. It also satisfies a relevant commutativity property, namely that the solution to the goal $q(X,Y), X = Y$ is the same as the solution to $q(X,Y)$, where each clause $q(X,Y) \leftarrow B$ is replaced by $q(X,Y) \leftarrow X = Y, B$. These are informal arguments, but we also note that the goal-independent analysis yields the *least*, that is, the most precise, model for the given pre-interpretation, which provides support for our claim that domains based on pre-interpretations are condensing.

3 Deriving a Pre-Interpretation from Regular Types

As mentioned above, a pre-interpretation of a language signature Σ is equivalent to a complete bottom-up deterministic FTA over Σ. An arbitrary FTA can be transformed to an equivalent complete, bottom-up deterministic FTA. Hence, we can construct a pre-interpretation starting from an arbitrary FTA.

An algorithm for transforming a non-deterministic FTA (NFTA) to a deterministic FTA (DFTA) is presented in [4]. The algorithm is shown in a slightly modified version.

input: NFTA $R = \langle Q, Q_f, \Sigma, \Delta \rangle$,
Set Q_d to $\emptyset$; set Δ_d to $\emptyset$
repeat
Set Q_d to $Q_d \cup \{s\}$, Δ_d to $\Delta_d \cup \{f(s_1, \ldots, s_n) \rightarrow s\}$
where
$\forall f^n \in \Sigma, \forall s_1, \ldots, s_n \in Q_d, \mathcal{C} = s_1 \times ... \times s_n$
$s = \{q \in Q | \exists (q_1, \ldots, q_n) \in \mathcal{C}, f(q_1, \ldots, q_n) \rightarrow q \in \Delta\}$
until no rule can be added to Δ_d
Set Q_{d_f} to $\{s \in Q_d \mid s \cap Q_{d_f} \neq \emptyset\}$
output: DFTA $R_d = \langle Q_d, Q_{d_f}, \Sigma, \Delta_d \rangle$

Description: The algorithm transform the NFTA from one that operates on states, to one that operates on sets of states from the NFTA. In the DFTA, the output of the algorithm, all reachable states in the NFTA are contained in sets that make up the new states - these are contained in the set Q_d. A state in the NFTA *can* occur in more than state in the DFTA. Potentially every non-empty subset of the set of states of the NFTA can be a state of the DFTA.

The sets in Q_d and the new set of transitions, Δ_d, are generated in an iterative process. In an iteration of the process, a function f is chosen from Σ. Then a number of sets, $s_1, \ldots, s_n$ corresponding to the arity of f, is selected from Q_d - the same set can be chosen more than once. The cartesian product is then formed, $(s_1 \times \cdots \times s_n)$, and for each element in the cartesian product, $q_1, \ldots, q_n$, such that a transition $f(q_1, \ldots, q_n) \rightarrow q$ exists, q is added to a set s. When all elements in the cartesian product have been selected, the set s is added to Q_d if s is non-empty and not already in Q_d. A transition $f(s_1, \ldots, s_n) \rightarrow s$ is added to Δ_d if s is non-empty.

The algorithm terminates when Q_d is such that no new transitions are added. Initially Q_d is the empty set, so no set containing a state can be chosen from Q_d

and therefore only the constants (0-ary functions) can be selected on the first iteration.

Example 6. In Example 3 a non-deterministic FTA is shown; $\Sigma = \{[]^0, [\_ \mid \_]^2, 0^0\}$, $Q = \{list, listlist, any\}$, $\Delta = \Delta_{any} \cup \{[] \to list, [any \mid list] \to list, [] \to listlist, [list \mid listlist] \to listlist\}$.

A step by step application of the algorithm follows:

Step 1: $Q_d = \emptyset, \Delta_d = \emptyset$. Choose f as a constant, $f = []$. Now $s = \{q \in Q \mid [] \to q \in \Delta\} = \{any, list, listlist\}$. Add s to Q_d and the transition $[] \to \{any, list, listlist\}$ to Δ_d.

Step 2: Choose $f = 0$. Now $s = \{q \in Q \mid 0 \to q \in \Delta\} = \{any\}$. Add s to Q_d and the transition $0 \to \{any\}$ to Δ_d.

Step 3: Choose $f = [\_ \mid \_]$, $s_1 = s_2 = \{any, list, listlist\}$. Now $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any, list, listlist\}$. Add s to Q_d and the transition $[\{any, list, listlist\} \mid \{any, list, listlist\}] \to \{any, list, listlist\}$ to Δ_d.

Step 4: Choose $f = [\_ \mid \_]$, $s_1 = s_2 = \{any\}$. Now $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any\}$. Add s to Q_d and the transition $[\{any\} \mid \{any\}] \to \{any\}$ to Δ_d.

Step 5: Choose $f = [\_ \mid \_]$, $s_1 = \{any\}, s_2 = \{any, list, listlist\}$. Now $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any, list\}$. Add s to Q_d and the transition $[\{any\} \mid \{any, list, listlist\}] \to \{any, list\}$ to Δ_d.

Step 6: Choose $f = [\_ \mid \_]$, $s_1 = \{any, list, listlist\}, s_2 = \{any\}$. Now $s = \{q \in Q | \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any\}$. Add s to Q_d and the transition $[\{any, list, listlist\} | \{any\}] \to \{any\}$ to Δ_d.

Step 7 to 11: No new sets added to Q_d. New transitions added to Δ_d: $[\{any, list\} \mid \{any, list\}] \to \{any, list\}$, $[\{any, list\} \mid \{any, list, listlist\}] \to \{any, list, listlist\}$, $[\{any, list, listlist\} \mid \{any, list\}] \to \{any, list\}$, $[\{any\} \mid \{any, list\}] \to \{any, list\}$, $[\{any, list\} \mid \{any\}] \to \{any\}$.

The states of states Q_d and the transitions Δ_d in the resulting DFTA are equivalent to the states and transitions in Example 3. $q_1 = \{any, list, listlist\}$, $q_2 = \{any, list\}$ and finally $q_3 = \{any\}$.

In a naive implementation of the algorithm where every combination of arguments to the chosen f would have to be tested in each iteration, the complexity lies in forming and testing each element in the cartesian product, for every combination of states in Q_d. It is possible to estimate of the number of operations required in a single iteration of the process, where an operation is the steps necessary to determine whether $f(q_1, \ldots, q_n) \to q \in \Delta$. Since Δ is static, an operation on Σ can be considered to be of constant time. The number of operations can be estimated by the formula $\#op = (s * e)^a$, where s is the number of states in Q_d, e is the average number of elements in a single state in Q_d and a is the arity of the chosen f. Every time a state is added to Q_d, an iteration in the algorithm will require additional operations. The worst case is if the algorithm causes an exponential blow-up in the number of states[4].

Obtaining a Complete FTA: The determinization procedure does not return a complete FTA in general. We can complete it as outlined in Section 2, by adding

an extra state and corresponding transitions. Another way is to ensure that the input NFTA accepts every term. We can easily do this by adding the standard transitions Δ_{any} to the input NFTA. The output DFTA is then guaranteed to be complete.

The algorithm's efficiency can be improved by generating the new states, Q_d, before the new transitions, Δ_d, are generated. Each iteration in the naive algorithm will redo work from previous iterations, though only combinations containing a newly added state can result in new states. The transitions can be generated in one iteration if all states in Q_d are known.

The new states are formed based on the transitions in the NFTA. The NFTA does not change during the algorithm and a preprocessing of the NFTA can be used to determine, for a given f^n, which states from Q_d can possibly occur as arguments in transitions: those states in Q_d not containing a state from the NFTA that occurring as and argument of f cannot result in any new state being added to Q_d.

Experimental results using an optimised version of the above algorithm (to be described in detail in a forthcoming paper) show that the algorithm can handle automata with hundreds of transitions. Table 3 in Section 5 gives some experimental results.

4 Examples

In this section we look at examples involving both types and modes. The usefulness of this approach in a binding time analysis (BTA) for offline partial evaluation will be shown. We also illustrate the applicability of the domains to model-checking.

We assume that Σ includes one special constant v (see Section 2.1). The standard type *any* is assumed where necessary (see Example 3), and it includes the rule $v \rightarrow any$.

Definition of Modes as Regular Types Instantiation modes can be coded as regular types. In other words, we claim that *modes are regular types*, and that this gives some new insight into the relation between modes and types. The set of ground terms over a given signature, for example, can be described using regular types, as can the set of non-ground terms, the set of variables, and the set of non-variable terms. The definition of the types *ground* (g) and *variable* (var) are $g = 0; []; [g|g]; s(g)$ and $var = v$ respectively. Using the determinization

Input states	Output states	Corresponding modes
g, var, any	{any,g}, {any,var}, {any}	ground, variable, non-ground-non-variable
g, any	{any,g}, {any}	ground, non-ground
var, any	{any,var}, {any}	variable, non-variable

Fig. 1. Mode pre-interpretations obtained from g, var and any

algorithm, we can derive other modes automatically. For these examples we assume the signature $\Sigma = \{[], [\_|\_], s, 0\}$ with the usual arities, though clearly the definitions can be constructed for any signature. Different pre-interpretations are obtained by taking one or both of the modes g and var along with the type any, and then determinizing. The choices are summarised in Figure 1. We do not show the transitions, due to lack of space. To give one example, the mode *non-variable* in the determinized FTA computed from var and any is given by the transitions for $\{any\}$.

$$\{any\} = 0; []; [\{any\}|\{any\}]; [\{any, var\}|\{any\}]; [\{any\}|\{any, var\}];$$
$$[\{any, var\}|\{any, var\}]; s(\{any\}); s(\{any, var\})$$

Let P be the naive reverse program shown below.

$$rev([], []). \qquad rev([X|U], W) \leftarrow rev(U, V), app(V, [X], W).$$
$$app([], Y, Y). \qquad app([X|U], V, [X|W]) \leftarrow app(U, V, W).$$

The result of computing the least model of P is summarised in Figure 2, with the abbreviations ground=g, variable=v, non-ground=ng, non-variable=nv and non-ground-non-variable=$ngnv$. An atom containing a variable X in the abstract model is an abbreviation for the collection of atoms obtained by replacing X by any element of the abstract domain. The analysis based on g and any is

Input types	Model
g, v, any	$\{rev(g,g), rev(ngnv, ngnv), app(g, var, ngnv), app(g, var, var),$ $app(g,g,g), app(g, ngnv, ngnv), app(ngnv, X, ngnv)\}$
g, any	$\{rev(g,g), rev(ng, ng), app(g, X, X), app(ng, X, ng)\}$
var, any	$\{rev(nv, nv), app(nv, X, X), app(nv, X, nv)\}$

Fig. 2. Abstract Models of Naive Reverse program

equivalent to the well-known Pos abstract domain [10], while that based on g, var and any is the **fgi** domain discussed in [7]. The presence of var in an argument indicates possible freeness, or alternatively, the absence of var indicates definite non-freeness. For example, the answers for rev are definitely not free, the first argument of app is not free, and if the second argument of app is not free then neither is the third.

Combining Modes with Other Types Consider the usual definition of lists, namely $list = []; [any|list]$. Now compute the pre-interpretation derived from the types $list$, any and g. Note that $list$, any and g intersect. The set of disjoint types is $\{\{any, ground\}, \{any, list\}, \{any, ground, list\}, \{any\}\}$ (abbreviated as $\{g, ngl, gl, ngnl\}$ corresponding to ground non-lists, non-ground lists, ground

lists, and non-ground-non-lists respectively). The abstract model with respect to the pre-interpretation is

$$\{rev(gl, gl), rev(ngl, ngl),$$
$$app(gl, X, X), app(ngl, ngnl, ngnl), app(ngl, gl, ngl), app(ngl, ngl, ngl)\}$$

Types for Binding Time Analysis Binding time analysis (BTA) for offline partial evaluation in LOGEN [11] distinguishes between various kinds of term instantiations. *Static* corresponds to *ground*, and *dynamic* to *any*. In addition LOGEN has the binding type *nonvar* and user-defined types.

A given set of user types can be determinized together with types representing *static*, *dynamic* (that is, *g* and *any*) and *var*. *Call types* can be computed from the abstract model over the resulting pre-interpretation, for example using a query-answer transformation (magic sets). This is a standard approach to deriving call patterns; [12] gives a clear account and implementation strategy.

Let P be the following program for transposing a matrix.

$$\begin{array}{ll}
transpose(Xs, []) \leftarrow & makerow([], [], []). \\
\quad nullrows(Xs). & makerow([[X|Xs]|Ys], [X|Xs1], [Xs|Zs]) \leftarrow \\
transpose(Xs, [Y|Ys]) \leftarrow & \quad makerow(Ys, Xs1, Zs). \\
\quad makerow(Xs, Y, Zs), & nullrows([]). \\
\quad transpose(Zs, Ys). & nullrows([[]|Ns]) \leftarrow nullrows(Ns).
\end{array}$$

Let *row* and *matrix* be defined as $row = []; [any|row]$ and $matrix = []; [row|matrix]$ respectively. These are combined with the standard types *g*, *var* and *any*. Given an initial call of the form $transpose(matrix, any)$, BTA with respect to the disjoint types results in the information that every call to the predicates *makerow* and *transpose* has a matrix as first argument. More specifically, it is derived to have a type $\{any, matrix, row, g\}$ or $\{any, matrix, row\}$, meaning that it is either a ground or non-ground matrix. Note that any term of type *matrix* is also of type *row*. This BTA is optimal for this set of types.

Infinite-State Model Checking The following example is from [13].

$$\begin{array}{lll}
gen([0,1]). & trans1([0,1|T],[1,0|T]). & trans(X,Y) \leftarrow \\
gen([0|X]) \leftarrow gen(X). & trans1([H|T],[H|T1]) \leftarrow & \quad trans1(X,Y). \\
reachable(X) \leftarrow & \quad trans1(T,T1). & trans([1|X],[0|Y]) \leftarrow \\
\quad gen(X). & trans2([0],[1]). & \quad trans2(T,T1). \\
reachable(X) \leftarrow & trans2([H|T],[H|T1]) \leftarrow & \\
\quad reachable(Y), trans(Y,X). & \quad trans2(X,Y). &
\end{array}$$

It is a simple model of a token ring transition system. A state of the system is a list of processes indicated by 0 and 1 where a 0 indicates a waiting process and a 1 indicates an active process. The initial state is defined by the predicate *gen* and the the predicate *reachable* defines the reachable states with respect to the transition predicate *trans*. The required property is that exactly one process is

active in any state. The state space is infinite, since the number of processes (the length of the lists) is unbounded. Hence finite model checking techniques do not suffice. The example was used in [14] to illustrate directional type inference for infinite-state model checking.

We define simple regular types defining the states. The set of "good" states in which there is exactly one 1 is *goodlist*. The type *zerolist* is the set of list of zeros. (Note that it is not necessary to give an explicit definition of a "bad" state).

$$\begin{array}{ll} one = 1 & goodlist = [zero|goodlist];\ [one|zerolist] \\ zero = 0 & zerolist = [];\ [zero|zerolist] \end{array}$$

Determinization of the given types along with *any* results in five states representing disjoint types: $\{any, one\}$, $\{any, zero\}$, the good lists $\{any, goodlist\}$, the lists of zeros $\{any, zerolist\}$ and all other terms $\{any\}$. We abbreviate these as *one*, *zero*, *goodlist*, *zerolist* and *other* respectively. The least model of the above program over this domain is as follows.

$$\begin{array}{ll} gen(goodlist) & trans1(goodlist, goodlist), trans1(other, other) \\ trans2(other, other) & trans(goodlist, goodlist), trans(other, other) \\ trans2(goodlist, other) & reachable(goodlist) \\ trans2(goodlist, goodlist) & \end{array}$$

The key property of the model is the presence of $reachable(goodlist)$ (and the absence of other atoms for *reachable*), indicating that if a state is reachable then it is a *goodlist*. Note that the transitions will handle *other* states, but in the context in which they are invoked, only *goodlist* states are propagated. In contrast to the use of set constraints or directional type inference to solve this problem, no goal-directed analysis is necessary. Thus there is no need to define an "unsafe" state and show that it is unreachable.

In summary, the examples show that accurate mode analysis can be performed, and that modes can be combined with arbitrary user defined types. Types can be used to prove properties expressible by regular types. Note that no assumption needs to be made that programs are well-typed; the programmer does not have to associate types with particular argument positions.

5 Implementation and Complexity Issues

The implementation is based on two components; the FTA *determinization* algorithm described in Section 3, which yields a pre-interpretation, and the computation of the least model of the program with respect to that pre-interpretation.

We have designed a much faster version of the determinization algorithm presented in Section 3. Clearly the worst-case number of states in the determinized FTA is exponential, but our algorithm exploits the structure of the given FTA to reduce the computation. Nevertheless the scalability of the determinization algorithm is a critical topic for future study and experiment. A forthcoming publication will describe our algorithm and its performance for typical FTAs. We

note that, although the states in the determinized FTA are formed from subsets of the powerset of the set of states in the input FTA, most of the subsets are empty in the examples we have examined. This is because there are many cases of subtypes and disjoint types among the given types.

The number of transitions in the determinized FTA can increase rapidly, even when the number of states does not, due to the fact that the output is a complete FTA. Hence, for each n-ary function, there are m^n transitions, if there are m states in the determinized automaton. We can alleviate the complexity greatly by making use of "don't care" arguments of functions in the transitions, of which there are usually several, especially in the transitions for the $\{any\}$ state, which represents terms that are not of any other type. If there exists an n-ary function f and states $q_1, \ldots, q_{j-1}, q_{j+1}, \ldots, q_n, q$ such that for all states q_j, there is a transition $f(q_1, \ldots, q_j, \ldots, q_n) \rightarrow q$, then we can represent all such transitions by the single transition $f(q_1, \ldots, q_{j-1}, X, q_{j+1}, , \ldots, q_n) \rightarrow q$. The j^{th} argument is called a *don't care* argument. Our algorithm generates the transitions of the determinized FTA with some "don't care" arguments (though not all the possible don't cares are generated in the current version), which is critical for the scalability of the model computation.

Abstract Compilation of a Pre-Interpretation The idea of abstract compilation was introduced first by Debray and Warren [15]. Operations on the abstract domain are coded as logic programs and added directly to the target program, which is then executed according to standard concrete semantics. The reason for this technique is to avoid some of the overhead of interpreting the abstract operations.

A pre-interpretation can be defined by a predicate $\rightarrow /2$ defining the FTA transitions. We introduce the predicate $\rightarrow /2$ directly into the program to be analysed, as follows. Each clause of the program of the form is transformed by repeatedly replacing non-variable terms occurring in the clause, of form $f(x_1, \ldots, x_m)$ where $x_1, \ldots, x_m$ $(m \geq 0)$ are variables, by a fresh variable u and adding the atom $f(x_1, \ldots, x_m) \rightarrow u$ to the clause body, until the only non-variables in the clause occur in the first argument of $\rightarrow$. If P is the original program, the transformed program is called $\bar{P}$.

When a specific pre-interpretation J is added to $\bar{P}$, the result is a *domain program* for J, called $\bar{P}^J$. Clearly $\bar{P}^J$ has a different language than P, since the definition of $\rightarrow /2$ contains elements of the domain of interpretation. It can easily be shown that least model $\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$ is obtained by computing $\mathsf{lfp}(T_{\bar{P}^J})$, and then restricting to the predicates in P (that is, omitting the predicate $\rightarrow /2$ which was introduced in the abstract compilation). An example of the domain program for *append* and the pre-interpretation for variable/non-variable is shown below. (Note that don't care arguments are used in the definition of $\rightarrow /2$).

$app(U, Y, Y) \leftarrow \quad [] \rightarrow U.$ $\qquad$ $app(U, Y, V) \leftarrow app(X, Y, Z), [X|X] \rightarrow U, [X|Z] \rightarrow V.$
$v \rightarrow var. \quad [] \rightarrow nonvar.$ $\qquad$ $[\_|\_] \rightarrow nonvar.$

Prog	Pre-int	NFTA		DFTA		Det.Time	Model time
P / $\|P\|$ / $\|\Sigma\|$	J	Q	Δ	Q_d	Δ_d	Secs	Secs
trans / 6 / 2	g,any	2	5	2	6	0.0	0.02
	var,any	2	4	2	7	0.0	0.02
	list,any	2	5	2	5	0.0	0.01
	matrix,row,any	3	7	3	8	0.0	0.03
	g,var,any	3	6	3	8	0.0	0.03
peep / 227 / 110	g,any	2	221	2	279	0.07	2.25
	var,any	2	112	2	349	0.05	1.29
	list,any	2	113	2	347	0.05	1.28
	matrix,row,any	3	115	3	515	0.05	1.98
	g,var,any	3	222	3	446	0.08	3.03
plan / 29 / 13	g,any	2	25	2	24	0.01	0.21
	var,any	2	14	2	30	0.01	0.11
	list,any	2	15	2	33	0.01	0.03
	matrix,row,any	3	17	3	34	0.0	0.13
	g,var,any	3	26	3	81	0.04	0.07
press / 155 / 32	g,any	2	66	2	58	0.03	0.92
	var,any	2	33	2	67	0.03	0.67
	list,any	2	34	2	65	0.02	0.66
	matrix,row,any	3	36	3	91	0.03	0.85
	g,var,any	3	64	3	83	0.03	1.01

Fig. 3. Experimental results

Computation of the Least Domain Model The computation of the least model is an iterative fixpoint algorithm. The iterations of the basic fixpoint algorithm, which terminates when a fixed point is found, can be decomposed into a sequence of smaller fixpoint computations, one for each *strongly connected component (SCC)* of the program's predicate dependency graph. These can be computed in linear time [16]. In addition to the SCC optimisation, our implementation incorporates a variant of the *semi-naive* optimisation [17], which makes use of the information about new results on each iteration. A clause body containing predicates whose models have not changed on some iteration need not be processed on the next iteration.

Experimental Results Figure 3 shows a few experimental results (space does not permit more). For each program, the table shows the number of clauses and the number of function symbols. The time to perform the determinization and compute the least model is shown. Timings were obtained using Ciao Prolog running on a machine with 4 Intel Xeon 2 GHz processors and 1 GByte of memory. The determinization algorithm currently does not find all the "don't care" arguments. Insertion of don't care values by hand indicates that the method scales better when this is done. More generally, finding efficient representations of sets of domain atoms is a critical factor in scalability. For two-element pre-interpretations such as Pos, BDDs [18] or multi-headed clauses [19] can be used.

6 Related Work and Conclusions

Prior work on propagating type information in logic programs goes back to [20] and [21]. Our work can be seen partly as extending and generalising the approach of Codish and Demoen [22]. Analysis of logic programs based on types was performed by Codish and Lagoon [23]. Their approach was similar in that given types were used to construct an abstract domain. However their types were quite restricted; each function symbol had to be of exactly one type (which is even more restrictive than top-down deterministic FTAs). Hence several of the application discussed in this paper are not possible, such as modes, or types such as the *goodlist* type of Example 4. On the other hand, their approach used a more complex abstract domain, using ACI unification to implement the domain operations, which allowed polymorphic dependencies to be derived. Like our approach, their domain was condensing.

Work on regular type inference is complementary to our method. The types used as input in this paper could be derived by a regular type inference, or set constraints. One possible use for the method of this paper would be to enhance the precision given by regular type inference. For example, (bottom-up) regular type inference derives the information that the first argument of $rev/2$ in the naive reverse program is a list; using a pre-interpretation derived from the inferred type, it can then be shown that the second argument is also a list. This approach could be used to add precision to regular type inference and set constraint analysis, which are already promising techniques in infinite state model-checking [14].

Applications in binding time analysis for offline partial evaluation have been investigated, with promising results. As noted in Section 4 various mode analyses can be reproduced with this approach, including Pos analysis [24].

References

1. Frühwirth, T., Shapiro, E., Vardi, M., Yardeni, E.: Logic programs as types for logic programs. In: Proceedings of the IEEE Symposium on Logic in Computer Science, Amsterdam. (1991)
2. Mishra, P.: Towards a theory of types in Prolog. In: Proceedings of the IEEE International Symposium on Logic Programming. (1984)
3. Yardeni, E., Shapiro, E.: A type system for logic programs. Journal of Logic Programming **10(2)** (1990) 125–154
4. 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)
5. Boulanger, D., Bruynooghe, M., Denecker, M.: Abstracting *s*-semantics using a model-theoretic approach. In Hermenegildo, M., Penjam, J., eds.: Proc. 6$^{\text{th}}$ International Symposium on Programming Language Implementation and Logic Programming, PLILP'94. Volume 844 of Springer-Verlag Lecture Notes in Computer Science. (1994) 432–446
6. Boulanger, D., Bruynooghe, M.: A systematic construction of abstract domains. In Le Charlier, B., ed.: Proc. First International Static Analysis Symposium, SAS'94. Volume 864 of Springer-Verlag Lecture Notes in Computer Science. (1994) 61–77

7. Gallagher, J., 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
8. Lloyd, J.: Foundations of Logic Programming: 2nd Edition. Springer-Verlag (1987)
9. Clark, K.: Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, London, Department of Computing (1979)
10. Marriott, K., Søndergaard, H.: Bottom-up abstract interpretation of logic programs. In: Proceedings of the Fifth International Conference and Symposium on Logic Programming, Washington. (1988)
11. Leuschel, M., Jørgensen, J.: Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. Elec. Notes Theor. Comp. Sci. **30(2)** (1999)
12. Codish, M., Demoen, B.: Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In Miller, D., ed.: Proceedings of the 1993 International Symposium on Logic Programming, Vancouver, MIT Press (1993)
13. Roychoudhury, A., Kumar, K.N., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A.: Verification of parameterized systems using logic program transformations. In Graf, S., Schwartzbach, M.I., eds.: Tools and Algorithms for Construction and Analysis of Systems, 6th Int. Conf., TACAS 2000. Volume 1785 of Springer-Verlag Lecture Notes in Computer Science. (2000) 172–187
14. Charatonik, W.: Directional type checking for logic programs: Beyond discriminative types. In Smolka, G., ed.: Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000. Volume 1782 of Springer-Verlag Lecture Notes in Computer Science. (2000) 72–87
15. Debray, S., Warren, D.: Automatic mode inference for logic programs. Journal of Logic Programming **5(3)** (1988) 207–229
16. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal of Computing **1(2)** (1972) 146–160
17. Ullman, J.: Implementation of Logical Query Languages for Databases. ACM Transactions on Database Systems **10(3)** (1985)
18. Schachte, P.: Precise and Efficient Static Analysis of Logic Programs. PhD thesis, Dept. of Computer Science, The University of Melbourne, Australia (1999)
19. Howe, J.M., King, A.: Positive Boolean Functions as Multiheaded Clauses. In Codognet, P., ed.: International Conference on Logic Programming. Volume 2237 of LNCS. (2001) 120–134
20. Bruynooghe, M., Janssens, G.: An instance of abstract interpretation integrating type and mode inferencing. In Kowalski, R., Bowen, K., eds.: Proceedings of ICLP/SLP, MIT Press (1988) 669–683
21. Horiuchi, K., Kanamori, T.: Polymorphic type inference in prolog by abstract interpretation. In: Proc. 6th Conference on Logic Programming. Volume 315 of Springer-Verlag Lecture Notes in Computer Science. (1987) 195–214
22. Codish, M., Demoen, B.: Deriving type dependencies for logic programs using multiple incarnations of Prop. In Le Charlier, B., ed.: Proceedings of SAS'94, Namur, Belgium. Volume 864 of Springer-Verlag Lecture Notes in Computer Science. (1994) 281–296
23. Codish, M., Lagoon, V.: Type dependencies for logic programs using ACI-unification. Theoretical Computer Science **238(1-2)** (2000) 131–159
24. Marriott, K., Søndergaard, H.: Precise and efficient groundness analysis for logic programs. LOPLAS **2(1-4)** (1993) 181–196