The Def-inite Approach to Dependency Analysis

Samir Genaim and Michael Codish

The Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
{genaim,mcodish}@cs.bgu.ac.il

Abstract. We propose a new representation for the domain of Definite Boolean functions. The key idea is to view the set of models of a Boolean function as an incidence relation between variables and models. This enables two dual representations: the usual one, in terms of models, specifying which variables they contain; and the other in terms of variables, specifying which models contain them. We adopt the dual representation which provides a clean theoretical basis for the definition of efficient operations on Def in terms of classic ACI1 unification theory. Our approach illustrates in an interesting way the relation of Def to the well-known set-Sharing domain which can also be represented in terms of sets of models and ACI1 unification. From the practical side, a prototype implementation provides promising results which indicate that this representation supports efficient groundness analysis using Def formula. Moreover, widening on this representation is easily defined.

1 Introduction

Boolean functions play an important role in various formal methods for specification, verification and analysis of software systems. In program analysis, Boolean functions are often used to approximate properties of the set of states encountered at a given program point. For example, a conjunction $x \wedge y$ could specify that variables x and y satisfy some property whenever control reaches a given program point. A Boolean function $\varphi_1 \rightarrow \varphi_2$ could specify that if φ_1 is satisfied at a program point (perhaps depending on the unknown inputs to the program) then also φ_2 is satisfied. A disjunction $\varphi_1 \vee \varphi_2$ could arise as a consequence of a branch in the control where φ_1 and φ_2 approximate properties of the `then` and `else` branches respectively.

For program analysis, we often consider the *positive* Boolean functions, Pos. Namely, those for which $f(true, \ldots, true) = true$. This restriction is natural as, due to the element of approximation, the result of an analysis is not a "*yes/no*" answer, but rather a "*yes/maybe not*" answer. In this case there is no "negative" information. Sophisticated Pos-based analyzers implemented using binary decision diagrams (BDD's) [4] have been shown [12] to give good experimental results both with regards to precision as well as with regards for the efficiency of the analyzers. However, scalability is a problem and inputs (programs) for which

D. Sands (Ed.): ESOP 2001, LNCS 2028, pp. 417–431, 2001.

the analysis requires an exponential number of iterations or exponentially large data structures are encountered [6].

In general, scalability problems with program analyses are often tackled by introducing widening operations [14]. Widening techniques can be applied to restrict the number of iterations as well as to limit the size of data-structures. A disadvantage of the use of binary decision diagrams for Pos-based groundness analyses is the lack of suitable widening techniques which maintain reasonable precision on the one hand and guarantee scalability on the other.

The domain, Def, of definite Boolean functions is a subdomain of Pos. These are the positive functions whose sets of models are closed under intersection. The domain Def is less expressive than Pos. For example, the formula $x \vee y$ is not in Def. However, Def-based analyzers can be implemented using less complex data structures and can be faster than Pos-based analyzers. As for precision, Def has been shown to provide a reasonable tradeoff between efficiency and precision for goal dependent groundness analyses (where a description of the inputs to the program being analyzed is given) [20,19]. Our present work indicates that in practice, for a large set of benchmark programs, goal independent Def analysis is just as precise as goal dependent analysis with respect to the groundness information derived. However, scalability remains a problem for Def. In [17], the authors demonstrate a series of inputs (programs) for which Def-based groundness analysis involves $2^{\sqrt{n}}$ iterations (where n is the size of the input). An advantage of basing analyses on Def is the fact that widening operations are not difficult to define and implement.

This paper investigates a new representation for Def functions. The key idea is to abstract the terms to the set of variables they contain, then dependencies between the terms are represented by set expressions. This turns out to be equivalent to viewing the models of a Boolean function (the dependencies) as an incidence relation between the variables of the function and its models which are sets of variables. This leads to two dual views of the structure. The usual view looks at the sets and specifies which are their elements. The dual view looks at the variables and specifies which are their sets. We take the dual view and show that Def is isomorphic to the domain of tuples of sets of variables modulo ACI1 equivalence.

The approach and the design of our domain build on two previous results, presented in [8] and in [9,10]. The work presented in [8] illustrates the application of logic programming with sets (of variables) to support an isomorphic representation for set sharing analysis. This approach is theoretically pleasing as the operations for sharing analysis are based on a classic ACI1 equality theory. In [9,10], the authors present an isomorphism between Sharing and Pos which leads to a better understanding of the similarities between groundness and sharing analyses. In particular Def analyses in the two domains are shown to coincide exactly extending an important result by Cortesi *et al.* [11]. From the practical point of view, the consequence is that the operations in a Sharing analysis are simplified when the focus is restricted to Def information.

This result is applied by King *et al.* in [20] where the authors quotient the Sharing domain to perform groundness analysis in Def. Their analyzer is reported to perform well in comparison to a Pos analysis using BDD's. In a sequel to [20] described in [19], the authors obtain a much faster analysis by factoring the representation into three components: G (ground arguments), E (equivalent arguments) and P (a propositional Horn clause component). The G and the E components can be represented as an atom containing only variables and the constant *true*. For example, the atom $p(A, true, A, B, true)$ represents the Def function $(x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$ on five variables. The approach is similar to the GER representation for Pos described in [3].

This paper, supports the view that groundness dependencies in Def can be obtained efficiently by quotienting a Sharing analysis. We are motivated by the preliminary results described in [20] and challenged by the speed of the analyzer described in [19]. We argue that when adopting the set logic programming approach of [8] we obtain "for free" a factoring of the domain to consider separately ground arguments and equivalence between arguments. Moreover, we obtain a representation which is easy to implement and to widen.

From the technical point of view, the set logic programming approach is attractive because we prove that the quotient of Sharing for Def in our representation is the well-studied and classic notion of ACI1 equivalence. Namely, two atoms with sets of variables as terms, representing corresponding Sharing elements are equivalent in Def if and only if they are ACI1 equivalent. All of the operations for Def analysis in Sharing are shown to be classic operations for ACI1 terms which are in our special case known to be efficient to maintain. This is in contrast to the use of ACI1 theory in the context of sharing analysis which requires a specialized notion of ACI1 unification (which chooses the most general unifier according to a non-standard ordering).

So, the quotient of Sharing for Def in our representation sums up to applying the standard ACI1 ordering on tuples of sets instead of the specialized ordering used in [8].

2 A Motivating Example

Groundness dependencies, as expressed using Boolean functions, capture information about the sets of variables that can occur in the terms the arguments of a predicate are bound to during the execution of a program. For example, the dependency $p(x_1, x_2) \leftarrow (x_1 \rightarrow x_2)$ specifies that if x_1 and x_2 are bound (during program execution) to the terms t_1 and t_2 respectively, then for any substitution θ, if θ grounds t_1 then it also grounds t_2. Essentially all this means is that the variables in t_2 are a subset of those in t_1. Viewing x_1 and x_2 as denoting sets of variables (instead of terms) we can represent this dependency as $p(x_1, x_2) \leftarrow (x_1 \supseteq x_2)$ or alternatively using *"set expressions"* as $p(x_1, x_2) \leftarrow (x_1 = a \cup b) \wedge (x_2 = b)$ where a and b are existentially quantified (names for) sets. This paper illustrates that representing groundness dependencies in terms of set expressions has practical implications.

Table 1 illustrates these two representations for positive Boolean functions and some relations between them. Each row in the first and second columns depicts a (positive) Boolean function φ on n variables V and its set of models $[\![\varphi]\!]$. Each model is represented as a set of variables indicating that the function assumes the value true under the assignment of these variables to *true* and all other variables to $false$. The model $\{x, y\}$ (for example, in the first row) is written xy for short and $\bot$ (for example, in the third row) denotes the empty model. Note that a positive function always contains V as one of its models. Throughout the example we take $n = 2$. In the third column the models of φ are

Table 1. Representations of Positive Boolean functions on $\{x, y\}$.

φ	$[\![\varphi]\!]$	$[\![coneg(\varphi)]\!]$	Tuple_n	$\varphi\downarrow$
$x \wedge y$	$\{xy\}$	$\{\bot\}$	$\langle \bot, \bot \rangle$	$(x \leftrightarrow true) \wedge (y \leftrightarrow true)$
x	$\{x, xy\}$	$\{y, \bot\}$	$\langle \bot, a \rangle$	$\exists_a.((x \leftrightarrow true) \wedge (y \leftrightarrow a))$
$x \leftrightarrow y$	$\{\bot, xy\}$	$\{xy, \bot\}$	$\langle a, a \rangle$	$\exists_a.((x \leftrightarrow a) \wedge (y \leftrightarrow a))$
$x \rightarrow y$	$\{\bot, y, xy\}$	$\{xy, x, \bot\}$	$\langle ab, a \rangle$	$\exists_{a,b}.((x \leftrightarrow a \wedge b) \wedge (y \leftrightarrow a))$
$y \rightarrow x$	$\{\bot, x, xy\}$	$\{xy, y, \bot\}$	$\langle a, ab \rangle$	$\exists_{a,b}.((x \leftrightarrow a) \wedge (y \leftrightarrow a \wedge b))$
$x \vee y$	$\{x, y, xy\}$	$\{y, x, \bot\}$	$\langle a, b \rangle$	$\exists_{a,b}.((x \leftrightarrow b) \wedge (y \leftrightarrow a))$
$true$	$\{\bot, x, y, xy\}$	$\{xy, y, x, \bot\}$	$\langle ac, ab \rangle$	$\exists_{a,b,c}.((x \leftrightarrow a \wedge c) \wedge (y \leftrightarrow a \wedge b))$

pointwise complemented with respect to V. These are the models of the "dual negation" of φ which is denoted $coneg(\varphi)$. Some readers may observe that the elements of this column correspond to elements of the Sharing domain (which always contain $\bot$ as a model). Dual negation is discussed in [10] and shown to provide an isomorphism between Sharing and Pos. The forth column represents φ in terms of set expressions. The set-expression $a \cup b$ is written ab for short and the notation is thus an n-tuple of sets ("Tuple_n") of variables disjoint from V. To obtain this representation, the models of $coneg(\varphi)$ are first given arbitrary names $(a, b, c, \ldots)$. The i^{th} set in a tuple then specifies which models of $coneg(\varphi)$ contain the i^{th} variable of V. For example, if we name the models in

$$[\![coneg(x \rightarrow y)]\!] = \{xy, x, \bot\}$$

by a, b and c respectively, then the Tuple_n representation of $x \rightarrow y$ is $\langle ab, a \rangle$: ab in the first position because x occurs in both a and b; and a in the second position because y occurs only in a. This is the representation used in [8] which is shown to be isomorphic to the Sharing domain when viewed modulo an appropriate notion of equivalence. It is also the representation we work with in this paper where we make the observation that looking at the elements of this column modulo standard ACI1 equivalence gives precisely Def.

The fifth column in Table 1 illustrates the relation between the tuples of sets representation of a positive Boolean function φ and its strongest logical

consequence which is in Def (denoted $\varphi\downarrow$). All of the functions in the example, except for $x \vee y$, satisfy $\varphi\downarrow = \varphi$ because they are already in Def. For $x \vee y$ we have $(x \vee y)\downarrow = true$. In the sequel (1)we will observe that the tuples $\langle a, b\rangle$ and $\langle ac, ab\rangle$ corresponding to $x \vee y$ and $true$ are ACI1 equivalent.

The class of definite Boolean functions is a complete lattice ordered by implication. The meet of two functions is their conjunction (Def is closed under conjunction) and the join is obtained by closing their disjunction under intersection of models (Def is not closed under disjunction).

Adopting the tuples of sets notation for Def, the join operation is defined as pointwise union. For example, the joins of x and $x \leftrightarrow y$ and of x and y are obtained as: $\langle \bot, b\rangle \sqcup \langle a, a\rangle = \langle a, ab\rangle$ and $\langle \bot, b\rangle \sqcup \langle a, \bot\rangle = \langle a, b\rangle$ which correspond to $y \rightarrow x$ and $true$ respectively. The meet is defined in terms of ACI1 unification. For example the meet of $x \rightarrow y$ and $y \rightarrow x$ is obtained as the ACI1 unification of the corresponding (renamed apart) tuples $\langle ab, a\rangle$ and $\langle c, cd\rangle$. The unification involves solving the system of ACI1 equations $\{ab = c, a = cd\}$ and applying the result on the given tuples. The result is $\langle a', a'\rangle$ which corresponds to the formula $x \leftrightarrow y$. In the sequel we formalize these definitions.

3 Preliminaries

Boolean Functions

Let $B = \{true, false\}$. A Boolean function on $V = \{x_1, \ldots, x_n\}$ is a function $\varphi : B^n \rightarrow B$. An *interpretation* $\mu : V \rightarrow B$ is an assignment of truth values to the variables in V. An interpretation μ is a *model* for φ, denoted $\mu \models \varphi$, if $\varphi(\mu(x_1), \ldots, \mu(x_n)) = true$. We write an interpretation as the set of variables which are assigned to the value $true$. The set of models of φ is thus viewed as a set of sets of variables defined by $[\![\varphi]\!]_V = \left\{ \{x \in V \mid \mu(x) = true\} \,\middle|\, \mu \models \varphi \right\}$. Much of the time we will omit the subscript V as it will be clear from the context.

Let φ be a Boolean function on V. We say that φ is *positive* if $V \in [\![\varphi]\!]$, that is, $\varphi(true, \ldots, true) = true$. We say that φ is *down-closed* if $[\![\varphi]\!]$ is closed under intersection. We denote by $\varphi\downarrow$ the strongest logical consequence of φ which is down-closed. In other words, $[\![\varphi\downarrow]\!] = [\![\varphi]\!]\downarrow$ is the smallest set that contains $[\![\varphi]\!]$ and is closed under intersection.

The class of positive Boolean functions ordered by logical consequence is a complete lattice denoted as Pos (sometimes we write Pos_V). An element of Pos is *definite* if it is down-closed. The definite Boolean functions form a sub-lattice of Pos denoted by Def (sometimes we write Def_V).

The *dual* of a Boolean function is the function that results when the roles of *false* and *true* are interchanged. For any Boolean function φ we denote by $coneg(\varphi)$ the dual of the negation of φ (or, equivalently, the negation of its dual). In terms of the models of a function φ we have the following *pointwise complementation principle* [10] which states that to move from $[\![\varphi]\!] = \{m_1, \ldots, m_n\}$ to its dual negated counterpart, we should replace each element m_i by its complement:

$$[\![coneg(\varphi)]\!]_V = \{ V \setminus m \mid m \in [\![\varphi]\!] \}.$$

ACI1 Equivalence and Tuples of Sets

We assume a denumerable set of variables $\mathcal{V}$ and an alphabet $\Sigma = \{\oplus, \bot\}$ consisting of a binary function symbol $\oplus$ and a constant symbol $\bot$. Sets of variables from $\mathcal{V}$ are represented as elements of the term algebra $T(\Sigma, \mathcal{V})$ modulo an equality theory consisting of the following axioms:

$$\begin{array}{ll} (x \oplus y) \oplus z = x \oplus (y \oplus z) & (\textit{associativity}) \\ x \oplus y = y \oplus x & (\textit{commutativity}) \\ x \oplus x = x & (\textit{idempotence}) \\ x \oplus \bot = x & (\textit{unit element}) \end{array}$$

The corresponding equivalence relation on terms is denoted $\approx_{aci1}$. This notion of equivalence suggests that set expressions can be viewed as flat sets of variables. For example, the terms $x_1 \oplus x_2 \oplus x_3$, $x_1 \oplus x_2 \oplus x_3 \oplus \bot$, and $x_1 \oplus x_2 \oplus x_3 \oplus x_2$ can each be viewed as representing the set $\{x_1, x_2, x_3\}$ of three variables. The set of n-tuples of set expressions is denoted $\mathsf{Tuple}_n(\Sigma, \mathcal{V})$. Set atoms are entities of the form $p(\bar{\tau})$ where p/n is an n-ary predicate symbol and $\bar{\tau}$ is an n-tuple of set expressions.

Set substitutions are substitutions which map variables of $\mathcal{V}$ to set expressions from $T(\Sigma, \mathcal{V})$. The application of a set substitution μ to a syntactic object τ is defined as usual by replacing occurrences of each variable x in τ by the set expression $\mu(x)$. The standard operations on set substitutions such as projection and composition are also defined just as for usual substitutions.

A preorder $\preceq_{aci1}$ on tuples of set expressions (and other syntactic objects) is defined as usual so that $\tau_1 \preceq_{aci1} \tau_2$ if there exists a set substitution μ such that $\tau_1 =_{aci1} \mu(\tau_2)$. In this case we say that τ_1 is less instantiated (modulo ACI1) than τ_2. This preorder induces a corresponding equivalence relation $\approx_{aci1}$ on tuples and a partial order $\preceq_{aci1}$ on the equivalence classes. For each natural n, $\mathsf{Tuple}_n = (\mathsf{Tuple}_n(\Sigma, \mathcal{V})/_{\approx_{aci1}}, \preceq_{aci1})$ is a complete lattice. The join is defined by

$$\langle \tau_1, \ldots, \tau_n \rangle \sqcup \langle \tau'_1, \ldots, \tau'_n \rangle = \langle (\tau_1 \oplus \tau'_1), \ldots, (\tau_n \oplus \tau'_n) \rangle$$

and the meet is defined through ACI1 unification [2]. An ACI1 unifier for two tuples τ_1 and τ_2 is a set substitution μ such that $\mu(\tau_1) =_{aci1} \mu(\tau_2)$.

Example 1. Consider the tuples $\bar{\tau}_1 = \langle a \oplus c, a \oplus b \rangle$ and $\bar{\tau}_2 = \langle a', b' \rangle$. Their ACI1 equivalence is illustrated by the substitutions $\mu_1 = \{a \mapsto \bot, b \mapsto b', c \mapsto a'\}$ and $\mu_2 = \{a' \mapsto a \oplus c, b' \mapsto a \oplus b\}$.

In our case, the underlying alphabet contains only the two (interpreted) function symbols $\{\oplus, \bot\}$. In this case the unification problem is classified as *elementary* [2]. The elementary unification problem is trivial (there always exists a unifier which binds all variables to $\bot$), the problem is unitary (there always exists a unique most general unifier) and the unification algorithm is efficient. The unique most general unifier for two set expressions (in our special case) τ and τ' is a substitution μ which maps variables to sets of variables as follows:

$$\mu(X) = \left\{ Z_{ab} \;\middle|\; \begin{array}{c} (a,b) \in \mathit{vars}(\tau) \times \mathit{vars}(\tau') \\ \mathit{and} \quad X \in \{a, b\} \end{array} \right\}$$

where a fresh variable Z_{ab} from $\mathcal{V}$ is introduced for each pair (a, b).

Example 2. Let $\tau = w \oplus u$ and $\tau' = x \oplus y$. The most general ACI1 unifier of τ and τ' is obtained as:

$$\mu = \begin{Bmatrix} w \mapsto Z_{wx} \oplus Z_{wy},\ u \mapsto Z_{ux} \oplus Z_{uy}, \\ x \mapsto Z_{ux} \oplus Z_{wx},\ y \mapsto Z_{uy} \oplus Z_{wy} \end{Bmatrix}.$$

4 Boolean Functions as Tuples of Sets

Let $V \subseteq \mathcal{V}$ be a finite set of variables. A set $M = \{m_1, \ldots, m_k\}$ of subsets of V can be represented as an incidence structure $S = (V, \mathcal{M}, \rho)$ where $\mathcal{M} \subseteq \mathcal{V}$ is a set of variables, disjoint from V, corresponding to the (names of the) sets in M and $\rho \subseteq V \times \mathcal{M}$ is a relation which specifies which elements from V belong to which sets from M. If $(x, m) \in \rho$ then we say that x and m are *incident*. For $m \in \mathcal{M}$ and $x \in V$ we denote the elements of m by $(m) = \{y \in V \mid (y, m) \in \rho\}$ and the sets containing x by $(x) = \{q \in \mathcal{M} \mid (x, q) \in \rho\}$. For $V = \{x_1, \ldots, x_n\}$, we denote the n-tuple $\langle (x_1), \ldots, (x_n) \rangle$ by $\mathcal{T}(M)$.

Example 3. Consider the set $M = \{\bot, x, y, xy\}$ over $V = \{x, y\}$. Its representation as an incidence structure is illustrated in Figure 1.

$(a) = \{x\}$ $\quad (x) = \{a, b\}$
$(b) = \{x, y\}$ $\quad (y) = \{b, c\}$
$(c) = \{y\}$
$(d) = \emptyset$ $\quad \langle (x), (y) \rangle = \langle \{a, b\}, \{b, c\} \rangle$

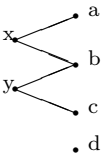

Fig. 1. Incidence structure for $\{\bot, x, y, xy\}$.

Viewing a set of sets of variables M over $V = \{x_1, \ldots, x_n\}$ as an incidence structure $(V, \mathcal{M}, \rho)$ provides two alternative representations. The original set M is obtained as $\{(m) \mid m \in \mathcal{M}\}$. A dual representation is obtained as $\mathcal{T}(M) = \langle (x_1), \ldots, (x_n) \rangle$. Note that the number of symbols in the two representations is always the same. Note also that the dual representation is ambiguous concerning the presence of the empty set. Luckily, in our application the empty set will always be present and hence there will be no ambiguity. This is because $\varphi \in \mathsf{Pos}$ if and only if $[\![coneg(\varphi)]\!]$ contains the empty set.

In the remainder of this section we prove that for $V = \{x_1, \ldots, x_n\}$, Def_V is isomorphic to Tuple_n.

Definition 1. *Let $V = \{x_1, \ldots, x_n\}$. We define the mapping iff : $\mathsf{Tuple}_n \to \mathsf{Def}_V$ such that for $\bar{\tau} = \langle \tau_1, \ldots, \tau_n \rangle \in \mathsf{Tuple}_n$:*

$$iff(\bar{\tau}) = \exists_{vars(\bar{\tau})} . \bigwedge_i (x_i \leftrightarrow \wedge \tau_i)$$

where $vars(\bar{\tau})$ denotes the set of variables in $\bar{\tau}$ and $\wedge\tau_i$ is the conjunction of the variables in the i^{th} argument τ_i of $\bar{\tau}$. Note that $iff(\bar{\tau}) \in \mathsf{Def}$ because Def is closed under existential quantification and $x_i \leftrightarrow (a_1 \wedge \cdots \wedge a_n)$ is equivalent to $[(a_1 \wedge \cdots \wedge a_n) \rightarrow x_i] \wedge [(x_i \rightarrow a_1) \wedge \cdots \wedge (x_i \rightarrow a_n)]$ which is in propositional definite form.

Example 4. Consider $\bar{\tau} = \langle a \oplus b, b \rangle$. We have $iff(\bar{\tau}) = \exists_{a,b}.(x_1 \leftrightarrow a \wedge b) \wedge (x_2 \leftrightarrow b)$ which is equivalent to $x_1 \rightarrow x_2$.

Theorem 1. *Let $V = \{x_1, \ldots, x_n\}$ then Tuple_n and Def_V are isomorphic domains.*

The results follows from Definition 1, and the following two lemmas which state that $\mathcal{T}$ is 1-1 and onto. The full proofs appear in [16].

Lemma 1. *Let $\tau_1, \tau_2 \in \mathsf{Tuple}_n$ then: $\tau_1 \leq_{aci1} \tau_2 \Leftrightarrow (iff(\tau_1) \rightarrow iff(\tau_2))$.*

Lemma 2. *Let $\varphi \in \mathsf{Pos}$ and $\bar{\tau} = \mathcal{T}([\![coneg(\varphi)]\!])$ then $iff(\bar{\tau}) = \varphi\downarrow$*

5 Groundness Analysis Using Tuple_n

The specification of a program analysis within the framework of abstract interpretation [13] requires: (1) a concrete semantics, (2) an abstract domain, and (3) abstract operations. Our abstract domain is Tuple_n and is isomorphic to Def. This section focuses on the abstract operations required to support groundness analyses as determined by the standard choices of a concrete semantics. In addition, we suggest two widening techniques which can be applied to improve the scalability properties of the analysis.

Representation and Abstract Operations

We adopt a non-ground representation. The sets of variables in a tuple are represented as lists of (Prolog) variables. In this way we benefit from the underlying renaming mechanism of Prolog. For example, the Def object $p(x_1, x_2) \leftarrow (x_1 \rightarrow x_2)$ is represented by $p([A, B], [A])$.

The isomorphism between Def and Tuple establishes the formal correspondence between the domain elements and operations (join and meet). The complexity of the join operator is linear in the number of the arguments (pointwise union). The complexity of the meet operator (ACI1 unification) is quadratic in the size of the atoms being unified (because ACI1 unification introduces a quadratic number of fresh variables). Projection is also straightforward to define (it is the same as the standard operation on atoms).

There is one additional operation that we must consider. Given two elements from Tuple_n (which are representatives of equivalence classes), we need to determine if they are ACI1 equivalent. To this end we introduce a *normal form*

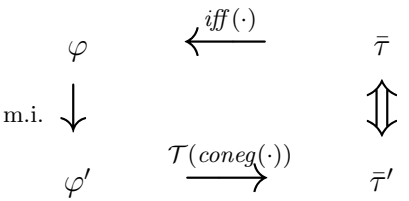

Fig. 2. Normal form representation in Tuple.

representation and hence an operation which given a tuple of sets computes its normal form.

Figure 2 illustrates the definition of the normal form operation which is more easily described through Def. We have a tuple of sets of variables $\bar{\tau}$ (on the top right side of the figure) with $iff(\bar{\tau}) = \varphi$ (on the top left). We take as φ' (bottom left) the conjunction of all of the Boolean functions $\psi \in \mathsf{Pos}$ such that $\psi{\downarrow} = \varphi$. Stated alternatively: $[\![\varphi]\!]'$ is the meet irreducible subset of $[\![\varphi]\!]$. Namely we remove from $[\![\varphi]\!]$ any model obtained as the intersection of other models. The existence and uniqueness of φ' is justified by Proposition 1 (below). Our normal form is the tuple (on the bottom right side of the figure) $\bar{\tau}' = \mathcal{T}(coneg(\varphi'))$.

Proposition 1. *Let* $\varphi_1, \varphi_2 \in \mathsf{Pos}$ *and* $\varphi_1{\downarrow} = \varphi_2{\downarrow}$ *then* $(\varphi_1 \wedge \varphi_2){\downarrow} = \varphi_1{\downarrow} = \varphi_2{\downarrow}$.

In the implementation, the normal form is computed directly on the tuple representation. Whereas in Def we removed sets obtained as the intersection of others, for tuples (which encode the *coneg* of the formula) we remove variables that correspond to models obtained as the union of others. The operation is described viewing the n-tuple in terms of the incidence structure it represents. We demonstrate the computation of the normal form of the tuple

$$\langle \underbrace{abce}_{x}, \underbrace{acd}_{y}, \underbrace{ae}_{z} \rangle$$

which represents the *coneg* models $\{a = \{x, y, z\}, b = \{x\}, c = \{x, y\}, d = \{y\}, e = \{z, x\}, f = \bot\}$. The corresponding incidence structure is illustrated in Figure 3 (on the left).

We should eliminate a (obtained as the union of e and d) and c (obtained as the union of d and b). We refer to the points on the left of the incidence structure (x,y,z) as *variables* and to those on the right (a,b,c,d,e) as *models*. The algorithm considers each model m to determine if can be obtained as the union of other models as follows:

1. color the variables connected to m;
2. color the models, except m, which are connected only to colored variables;
3. if each of the colored variables is connected to at least one colored model then remove m.

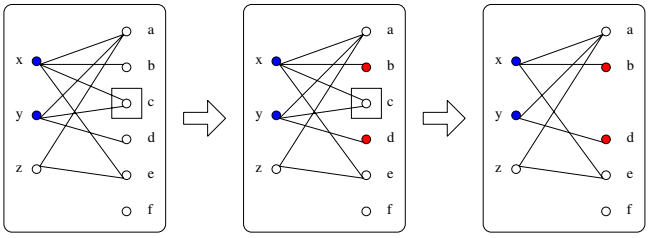

Fig. 3. Computing a normal form: eliminating c.

Figure 3 depicts the process of eliminating the model c in our example. First the variables x and y (connected to c) are colored and then the models b and d (connected only to colored variables) are colored. The colored variables are each connected to at least one colored model, so c is removed. Repeating this operation for all models gives the normal form $\langle be, d, de\rangle$.

The complexity of the algorithm is $O(|V| * |E|)$, where $|V|$ and $|E|$ are the number of vertices and edges in the incidence structure (note that $|V|$ and $|E|$ might be exponential in the number of arguments).

Widening the Tuples

The size of a (normal form) representation of a tuple of sets is potentially exponential in the number of sets. This can be problematic because the operations for meet and for computing a normal form (as implemented) are quadratic in the size of their inputs. As an illustration, consider the n-ary Boolean function whose set of models consist of the empty set together with the $\binom{n}{n/2}$ sets with $\frac{n}{2}$ variables. This set of models is meet irreducible and of exponential size. Hence so is the normal form of its dual representation as a tuple of sets.

In addition to the potential size problem, we also know from [17] that the number of iterations in a Def analysis has a worst case of at least $2^{\sqrt{n}}$. To guarantee a robust analysis we introduce two possible widening operations [14] on our tuple representation. Note that widening operations are usually used to avoid infinite chains during an analysis. In our case, the domain is of finite height. Widening is applied to avoid generating tuples with large sets as well as to reduce the number of iterations needed to reach a fixed point.

Restricting the size of sets: A simple widening operation is obtained by restricting the size of the sets which occur in a (normalized) tuple to contain at most k variables. Similar to a depth-k abstraction on terms [21], a tuple which violates the restriction is replaced by a more general tuple which contains smaller sets and captures less dependencies between arguments. In contrast to a depth-k abstraction on terms the operation on tuples of sets cannot be described as an abstraction in the adjoint framework [13] because there is no *"best approximation"*.

Restricting the size of the sets in a tuple limits the types of dependencies that we can express. For example, with $k = 1$ we can express groundness information and equivalences between arguments as in $p([A], [\,], [A], [B], [\,])$ which corresponds in Def to $p(x_1, x_2, x_3, x_4, x_5) \leftarrow (x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$. With $k = 2$ we can express simple dependencies such as in $p([A, B], [A])$ which corresponds in Def to $p(x_1, x_2) \leftarrow (x_1 \rightarrow x_2)$ and as $append([A], [B], [A, B])$ which corresponds in Def to $append(x_1, x_2, x_3) \leftarrow ((x_1 \wedge x_2) \leftrightarrow x_3)$.

The widening algorithm should generalize a given tuple $\bar{\tau}$ (which contains sets larger than k) to provide a tuple $\bar{\tau}'$ in which the sets have no more than k elements. It is always possible to find such a generalization because the most general tuple $\langle [\_], [\_], \ldots, [\_] \rangle$ contains sets of size 1 consisting of distinct variables.

A naive widening algorithm simply "drops" sets with more than k elements (replacing them by fresh singletons) hence maintaining the dependencies between the smaller sets.

Example 5 (Naive widening). Consider $\pi = p([A, B, C, D], [A, B, C], [A, B], [A])$, which corresponds in Def to $p(x_1, x_2, x_3, x_4) \leftarrow (x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_3) \wedge (x_3 \rightarrow x_4)$, and $k = 2$. The naive widening of π with $k = 2$ results in $p([X],[Y],[A,B],[A])$ which corresponds in Def to $p(x_1, x_2, x_3, x_4) \leftarrow (x_3 \rightarrow x_4)$. The dependencies (in π) that involve sets with more than 2 elements, i.e $(x_1 \rightarrow x_2)$ and $(x_2 \rightarrow x_3)$, are lost.

A less naive widening algorithm might capture some of the dependencies between the large sets. A simple approach is to rename the variables in the large sets and apply normalization. For instance, applying this technique to the tuple $p([A, B, C, D], [A, B, C], [A, B], [A])$ from Example 5 gives $p([X,W,Z,Y],[X,W,Z],[A,B],[A])$ which is normalized to $p([X, Y], [X], [A, B], [A])$. This approach may be applied iteratively until no sets are larger than k, or until normalization does not further reduce the size of the sets in the tuple (in which case we can apply the naive approach).

It is interesting to note that widening with $k = 1$ results in an analysis over the domain $EPos$ described in [18] which is a polynomial analysis that captures groundness information and equivalences between pairs of variables. The reader familiar with that work will note that where $EPos$ analysis is inefficient due to the use of non-deterministic iff/2 atoms, our approach applies deterministic ACI1 unifications. In [18] the authors avoided the non-determinism of $EPos$ by application of local iteration with some loss of precision. Our approach with $k = 1$ is both efficient and precise.

Restricting the size of unifications: An alternative approach to avoid generating large sets in tuples is to restrict the application of operations which generate complex dependencies. Given a sequence of ACI1 unifications that need to be solved (for example unifications of the atoms of a clause body with the current approximations for these atoms) we can obtain a more general solution is obtained by "dropping" unifications between large sets. Recall that the unification between sets containing n and m variables respectively generates $n * m$ fresh variables. If n and m are large this introduces an even larger number of

fresh variables, many of which are redundant and eliminated by a subsequent but costly application of the normalization algorithm. Strictly speaking this operation could be described as a less precise unification algorithm. But to formalize it that way we would have to normalize before each application of unification.

With this approach we drop equations such that both m and n (the number of variables in each side) are greater than a constant u. For example, with $u = 1$ we drop any unification for which both m and n are larger than 1. Note that the ACI1 unification for $u = 1$ is very efficient because: (1) Solving equations of the form $\bot = [A_1, \ldots, A_n]$ results in the bindings $\{A_1 \mapsto \bot, \ldots, A_n \mapsto \bot\}$; and (2) Solving equations of the form $[B] = [A_1, \ldots, A_n]$ gives $\{B \mapsto [A_1, \ldots, A_n]$. In both cases no fresh variables are introduced.

6 Experimentation

We have implemented a prototype analyzers based on the Tuple domain isomorphic to Def for groundness analysis. The implementation is based on a simple bottom-up meta-interpreter enhanced by ACI1 unification. The implementation was coded in SICStus Prolog (version 3, release 7), and the experiments where performed on a Pentium III (300 MHz) and 64MB memory that run Linux Redhat 6.2 (kernel 2.2.4-15). The implementation effort itself involved a small effort given the analyzers described in [8] and in [7]. The main difference is in the simplifications made in the unification and in the normal form algorithms.

For goal-dependent analyses we use an interpreter which applies induced magic-sets [5] and eager evaluation [22]. This is basically the same analysis engine used in [20] and [19]. We have performed 4 experiments for goal-dependent analysis, on a set of 60 benchmarks programs from the benchmark suit used in [19]. Analysis times are described in Table 2. The first two columns in the table ($\mathsf{Def}_{k=1}$ and $\mathsf{Def}_{k=2}$) correspond to Def analysis widened (using the less naive approach described above) to restrict the size of sets to $k = 1$ and $k = 2$. The third column ($\mathsf{Def}_{u=1}$) corresponds to a Def analysis restricting the size of unifications to $u = 1$. The last column corresponds to a Def analysis without widening. The entries of the first three columns represent the time (in seconds) required to perform the analysis. Programs from the benchmark suit, the Def analysis of which requires less than 0.1 seconds and which exhibit no loss of precision for the widenings, are not shown (the full table is available in [16]). In the first three columns the notation $(\frac{n_1}{n_2})$ indicates a loss of precision of n_1 ground arguments with respect to the n_2 ground arguments found in the corresponding Def analysis.

For Def analyses the analyzer exhibits better timings than those described in [20] (for Def and widenings of Def). It *"times out"* only on the `aqua_c` benchmark, while the analyzer of [20] *"times out"* also on `reducer.pl` and `ili.pl`. For $\mathsf{Def}_{k=1}$ the analyzer looses some precision on 6 out of the 60 programs. This analysis is equivalent to groundness analysis over the $EPos$ domain that described in [18]. Our analyzer is faster than the non-deterministic version of [18]. For $\mathsf{Def}_{k=2}$ the analysis looses some precision for three benchmark programs. For $\mathsf{Def}_{u=1}$ the analysis gives no lose of precision Comparing to the timing results of our analyzer to those of [19] show that their analyzer is 2 to 5 times faster than

Table 2. Benchmarks : Times (in seconds) and loss of precision.

Program	$\mathsf{Def}_{k=1}$	$\mathsf{Def}_{k=2}$	$\mathsf{Def}_{u=1}$	Def
aircraft	0.75	0.73	0.70	0.69
ann	0.33	0.49	0.41	0.52
asm	0.15	0.15	0.15	0.15
bryant	0.27	0.49	0.39	0.89
chat_80	2.30 ($\frac{3}{855}$)	2.86 ($\frac{3}{855}$)	2.85	3.38
chat_parser	0.96 ($\frac{1}{505}$)	0.99 ($\frac{1}{505}$)	1.16	1.18
essln	0.39 ($\frac{4}{162}$)	0.39	0.39	0.40
flatten	0.13	0.16	0.17	0.22
ili	0.40	0.57	0.54	1.15
ime_v2-2-1	0.12 ($\frac{1}{101}$)	0.16	0.17	0.17
lnprolog	0.21 ($\frac{33}{145}$)	0.18	0.17	0.17
map	0.34	0.34	0.33	0.32
music	0.07	0.10	0.11	1.34
nandc	0.05 ($\frac{3}{37}$)	0.05	0.04	0.05
nbody	0.14	0.16	0.16	0.16
peep	0.15	0.18	0.19	0.18
peval	0.36	0.42	0.44	0.59
press	0.28 ($\frac{1}{53}$)	0.42	0.36	4.59
reducer	0.22	0.29	0.31	3.77
rotate	0.00 ($\frac{1}{3}$)	0.01	0.00	0.01
rubik	0.26	0.24	0.24	0.23
scc1	0.25	0.26	0.24	0.25
sdda	0.14	0.16	0.17	0.18
semi	0.14	0.16	0.16	0.15
sim	0.64	0.93	0.87	0.99
sim_v5-2	0.15	0.15	0.14	0.33
simple_ana	0.54	0.66	0.74	1.70
trs	0.47	0.72	0.63	0.78
unify	0.16	0.19	0.23	0.24
aqua_c	14.55 ($\frac{63}{1285}$)	86.41 ($\frac{33}{1285}$)	21.16	time out

our analyzer and in particular it does not *"times out"* on the `aqua_c` benchmark. This is mainly because of the technique they apply for *"entailment checking"* (checking if a new atom descreption is already entailed by an old one without applying join and projection). Experiments show that in our analyzer more than 80% of the attempts to add new tuples to the database fails. Namely, more that 80% of the calls to join and normal form could be avoided if we had *"entailment checking"*.

We also implemented a goal-independent analyzer based on a semi-naive interpreter optimized for strongly connected components. For this analyses, it is interesting to note that although Def is known not to be a condensing domain, the amount of groundness obtained when applying a goal independent analysis is precisely the same as with the goal dependent analysis for the set of benchmarks chosen. However, for goal independent analyses, extreme widening such as $k = 2$ and $u = 1$ are not realistic with regards to the loss of precision. A table with run times for this analysis available in [16].

7 Conclusion

The study of the relationships between Pos, Sharing and Def has been the topic of many research papers in the last 10 years. Analyses using Def were defined early on in the work of Dart [15] and an evaluation of various representations for Def can be found in [1]. However, the use of BDD's for Pos analysis is more popular. Due to the difficulty in widening BDD representations, it is a recent trend to look for weaker domains which support analyses that are fast and do not exhibit (or can be widened so as not to exhibit) the underlying worst case complexity for the occasional hard example. Recent results described in [20] and [19] indicate that Def is a promising candidate for the groundness analysis of (constraint) logic programs.

We provide a new representation for Def based on n-tuples of sets of variables as proposed in [8] for representing set Sharing. Our main result states that the quotient of Sharing to Def is precisely the quotient of our n-tuples representation with respect to ACI1 equivalence. From the practical point of view this provides the basis to implement program analysis over Def using the classic and well-studied ACI1 unification algorithms. For the special case where the only constant symbol is the unit element this is efficient and simple to implement. An advantage of basing Def analyses on tuples representation is the simplicity of which the widening can be defined and implemented.

The absence of an *"entailment checking"* for tuples is the main reason why the analysis of [19] is faster than ours. Filling this gap is the topic of ongoing reaserch.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 41–126. Oxford Science Publications, 1994.
3. R. Bagnara and P. Schachte. Factorizing equivalent variable pairs in ROBDD-based implementations of *Pos*. In A. M. Haeberer, editor, *AMAST'98*, volume 1548 of *LNCS*, pages 471–485, Amazonia, Brazil, 1999. Springer-Verlag, Berlin.
4. R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
5. M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):354–370, 1999.
6. M. Codish. Worst-case groundness analysis using positive Boolean functions. *Journal of Logic Programming*, 41(1):125–128, 1999.
7. M. Codish and V. Lagoon. Type dependencies for logic programs using aci-unification. *Theoretical Computer Science*, 2000.
8. M. Codish, V. Lagoon, and Bueno F. An algebraic approach to sharing analysis of logic programs. *Journal of Logic Programming*, 41(2):110–149, 2000.
9. M. Codish and H. Søndergaard. The Boolean logic of set sharing analysis. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *LNCS*, pages 89–101, Berlin, 1998. Springer.

10. M. Codish, H. Søndergaard, and P. J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
11. A. Cortesi, G. Filé, and W. Winsborough. The quotient of an abstract interpretation. *Theoretical Computer Science*, 202(1–2):163–192, 1998.
12. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Evaluation of the domain Prop. *Journal of Logic Programming*, 23(3):237–278, 1995.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
14. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. Number 631 in LNCS, pages 269–295, Leuven, Belgium, 1992. Springer-Verlag, Berlin.
15. P. W. Dart. On derived dependencies and connected databases. *The Journal of Logic Programming*, 11(1 & 2):163–188, July 1991.
16. S. Genaim and M. Codish. The Def-inite Approach to Dependency Analysis. Technical report, Computer Science, Ben-Gurion University, 2000. http://www.cs.bgu.ac.il/~mcodish/Papers/ppapers.html.
17. S. Genaim, J.M. Howe, and M. Codish. Worst-case groundness analysis using definite boolean functions, September 2000.
18. A. Heaton, M. Abo-Zaed, M. Codish, and A. King. A Simple Polynomial Groundness Analysis for Logic Programs. *Journal of Logic Programming*, 45:143–156.
19. J. M. Howe and A. King. Implementing groundness analysis with definite Boolean functions. In G. Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2000.
20. A. King, J. G. Smaus, and P. Hill. Quotienting share for dependency analysis. In Doaitse Swierstra, editor, *ESOP*, volume 1576 of *LNCS*, pages 59–73. Springer-Verlag, April 1999.
21. Taisuke Sato and Hisao Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34(1–2):227–240, November 1984.
22. J. Wunderwald. Memoing evaluation by source-to-source transformation. In Maurizio Proietti, editor, *Proceedings of the Fifth International Workshop on Logic Program Synthesis and Transformation*, volume 1048 of *LNCS*, pages 17–32. Springer.