Abstract Interpretation Using TDGs

Laurent Mauborgne*

LIENS, École Normale Supérieure, 45 rue d'Ulm, 75230 Paris cedex 05, France

Abstract. This paper intends to give an efficient way of implementing abstract interpretations. The idea is to use a good symbolic representation of boolean functions, TDGs, a refinement of Binary Decision Diagrams. A general way of using this representation in abstract interpretation is given, in particular we examine the possibility of encoding higher order functions into TDGs. Moreover, this representation is used to design a widening operator based on the size of the objects represented, so that abstract interpretations will not fail due to insufficient memory. This approach is illustrated on strictness analysis of higher-order functions, showing a great increase of efficiency.
keywords: Abstract interpretation, BDD, Strictness analysis, Higher order, Practical implementation.

1 Introduction

Writing a program of some importance, one will soon need an automatic program analyzer, just to see if what is programmed is effectively what was intended. Such an analyzer would anyway be needed when one comes to optimization. The most important problem when designing a program analyzer is undecidability, the most well known being the termination problem. To handle this, the solution is to give partial or approximate answers. Abstract interpretation is the theoretical framework to design automatic program analysis, allowing sound approximations. This theory works quite well, but quite often, when the analysis is too precise, it becomes unusable, because of the amount of memory, or the time it would take. As for every function programmed on a computer, this analysis can be reduced to a boolean function, or an array of boolean functions. So the idea here is to combine the possibility of approximation of abstract interpretation with a good symbolic representation of boolean functions; that way, one could design an analyzer that would take less space and less time.

2 Typed Decision Graphs

Typed Decision Graphs [8], or TDGs, are powerful symbolic representations of boolean functions. They are a refinement of the well-known Binary Decision Diagrams [5], or BDD, which are already wildly used in many fields, such as circuits synthesis and verification [11, 10, 14], or protocols verification [12, 13].

* email: Laurent.Mauborgne@ens.fr

Even if it is still not very used in abstract interpretation (but see [17]), this representation seems to be efficient enough to be worth using.

2.1 Formalization Work on Binary Decision Diagrams

First, we shall define what objects are encoded by BDDs: boolean functions with the names of the variables used to calculate them.

let Var be a totally ordered ($<^v$) set of variables. $Var_n \stackrel{\text{def}}{=} \{V \subseteq Var \mid |V| = n\}$. $\mathcal{B}_n \stackrel{\text{def}}{=} Var_n \times (\{0,1\}^n \to \{0,1\})$. $\mathcal{B} \stackrel{\text{def}}{=} \bigcup_n \mathcal{B}_n$. For simplicity, we will always order the indexes of set of variables following the order on var. So when we will write $\{x_1, \ldots, x_n\} \in Var_n$ it will mean $\forall i,\ 1 \leq i \leq n,\ x_i \in Var$ and $x_1 <^v \ldots <^v x_n$. We will note $f(x_1, \ldots, x_n)$ for $(\{x_1, \ldots, x_n\}, f)$. We'll define too $\mathcal{V}(f(x_1, \ldots, x_n)) \stackrel{\text{def}}{=} \{x_1, \ldots, x_n\}$.

BDDs are based on Shannon's trees, which are just decision trees, whose uniqueness is insured by Shannon's expansion theorem [1]. Expressed in our formalism, this theorem is:

Theorem 1 Shannon's expansion. *Let* $f(x_1, \ldots, x_n) \in \mathcal{B}_n$.
$\forall i,\ 1 \leq i \leq n,\ \exists!(f_{\overline{x_i}}, f_{x_i}) \in (\mathcal{B}_{n-1} \times \mathcal{B}_{n-1})$ *such that:*

$$f(x_1, \ldots, x_n) = \neg x_i \wedge f_{\overline{x_i}} \vee x_i \wedge f_{x_i}$$

A Shannon's tree is a binary tree labeled with variables, 0 or 1. A binary tree T can be defined as a partial function from $\{0,1\}^\star$, the set of all finite words on $\{0,1\}$, to the set of labels, with the prefix closure property: the domain is not empty, and if a word uv is in its domain, then u is in its domain too[2]. The Shannon's tree representing $f(x_1, \ldots, x_n)$ is defined as follows:

$$\begin{aligned} \mathrm{St}(f(x_1, \ldots, x_n))(u) \stackrel{\text{def}}{=}\ & \text{if } |u| < n \text{ then } x_{|u|+1} \\ & \text{if } u = a_1 a_2 \ldots a_n \text{ then } f(a_1, a_2, \ldots, a_n) \end{aligned}$$

where $|u|$ is the length of u.

BDDs are compact representation of Shannon's trees, obtained using two simple reduction rules: sharing and elimination.

Sharing. The first idea that comes, when one wants to reduce the size taken by a representation of a tree, is to share isomorphic subtrees. The tree will then become a directed acyclic graph. A binary decision DAG can be defined recursively as being either a node N of $Var \times bdd \times bdd$ or a leaf in $\{0,1\}$. Figure 1 shows how it is represented. Dashed lines represent the first or "left" subtree.

The transformation is described by the *share* function, so it is obvious that the representation is still unique.

$$share(St) \stackrel{\text{def}}{=} \text{if } St = root(k) \text{ then } k \text{ else } N(St(\epsilon), St\backslash 0, St\backslash 1)$$

[2] uv is the concatenation of u and v

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

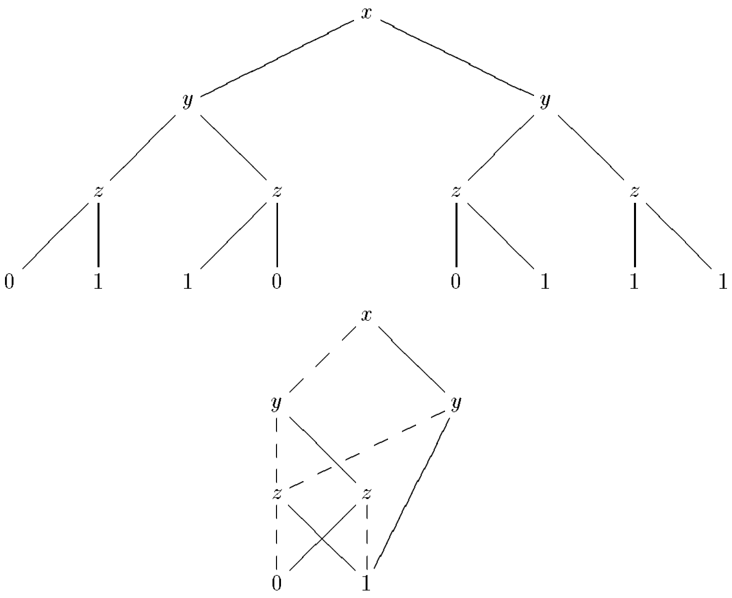

Fig. 1. Construction of a BDD.

where ϵ is the empty word, $root(k)$ is the tree with domain $\{\epsilon\}$ and value k, and $T\backslash u$ is the subtree of T with domain $dom(T\backslash u) \stackrel{\text{def}}{=} \{v|uv \in dom(T)\}$ and such that $T\backslash u(v) \stackrel{\text{def}}{=} T(uv)$.

The sharing comes from the fact that if two subtrees are isomorphic, then the mathematical objects representing the subtrees are equal, so the results of *share* on them are the same.

Elimination of Superfluous Nodes. What we see now, is that some nodes are totally unuseful from the point of view of a decision diagrams. So the next step is to suppress them.

Once again, the transformation can be written as transformation rules, so the representation is still unique.

$$supp(N(x, d_1, d_2)) = \text{ if } d_1 = d_2 \text{ then } supp(d_1) \text{ else } N(x, supp(d_1), supp(d_2))$$

The only problem is that doing so, a BDD does not only represent one function of $\mathcal{B}$, but all the functions whose result is the same regardless of the assignment of additional variables not appearing in the BDD. For example, if $\forall x, y, z,\ f(x, y, z) = g(y)$ then $f(x, y, z)$ is represented by the same BDD as $g(y)$. But quite often, having the same representations for such functions is not important, as they are often thought of as "equivalents"[3].

[3] In fact, they are the same representation of the function from $\{0, 1\}^\omega$ to $\{0, 1\}$.

2.2 TDGs

To gain again more room, the idea is to go back to Shannon's trees and to produce more isomorphic subtrees. Then we will apply the same reduction steps.

Typed Shannon's Trees. The idea of typed Shannon's trees [3] came from the remark that:

$$\neg f = \neg x \wedge \neg f_{\overline{x}} \vee x \wedge \neg f_x$$

This means that from the point of view of the Shannon's tree, f and $\neg f$ are identical, except for the leaves: 0 becoming 1 and 1 becoming 0. So instead of having two different trees, why not having only two different signs. Typed Shannon's trees are just that, trees with signs. To be more precise, the labeling set becomes $\{-, +\} \times (Var \bigcup \{0, 1\})$. And if T such that $T(\epsilon) = (s, l)$ represents f then changing just the value of T on ϵ such that it becomes $(-s, l)$ makes it represent $\neg f$.

Now the problem is that by using simple Shannon's trees and just adding signs, canonicity is lost: 0 can be represented by $(+, 0)$ or $(-, 1)$ for example. Well, we simply make a choice, once for all. The first that could come to mind is simply take the Shannon's tree, and add + everywhere. Well, there the typing is not *that* useful. Here is one that gives more results [9]:

$$\begin{aligned}\mathrm{Tst}(f(x_1, \ldots, x_n))(a_1 \ldots a_i) \stackrel{\text{def}}{=} &\text{ if } f(a_1, \ldots, a_i, 1, \ldots, 1) = 1 \\ &\text{ then } (+, \mathrm{St}(f(x_1, \ldots, x_n))(a_1 \ldots a_i)) \\ &\text{ else } (-, \mathrm{St}(\neg f(x_1, \ldots, x_n))(a_1 \ldots a_i))\end{aligned}$$

The resulting tree is represented on fig. 2, where the signs have been put on the edges instead of the labels, and only minus have been represented to give a more compact representation.

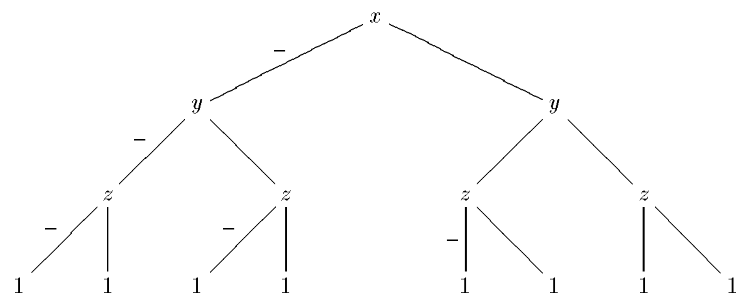

Fig. 2. Typed Shannon's tree.

Resulting Graph. Now, we just apply the same reduction rules as for a BDD, still assuring the uniqueness of the representation, and we've got Typed Decision Graphs. To have the value of the function for a given assignment, follow the same method as for BDD, counting the number of $-$ in the path. If this number is odd then the result is 0, if it is even, 1.

Fig. 3. The TDG for $f(x, y, z)$, and $f(y, x, z)$.

The size of the TDGs seems quite reasonable, and it is in most case, but there are still cases where it is exponential in the number of its variables [15]. If we restrict the representation to boolean functions without explicit variables, then it is sometime possible to reduce the size of the TDG representing the function by changing the order of the variables[4], but there are still cases where the representation is exponential whatever the order on variables.

2.3 Operators on TDGs

Not only does symbolic representation save space, but it saves time too, assuming the operators on boolean functions are correctly translated.

An operator is a function from $\mathcal{B}^n$ to $\mathcal{B}$. The key property that leads to a fast computation of operators is orthogonality.

Definition 2. Let Op be a n-operator. Op is said orthogonal iff:

$$\begin{cases} \forall f_1, \ldots, f_n \in \mathcal{B}, \forall x \in Var, \\ \quad Op(f_1, \ldots, f_n) = \neg x \wedge Op(f_{1\overline{x}}, \ldots, f_{n\overline{x}}) \vee x \wedge Op(f_{1x}, \ldots, f_{nx}) \\ \forall k_1, \ldots, k_n \in \mathcal{B}_0, Op(k_1, \ldots, k_n) \in \mathcal{B}_0 \end{cases}$$

For example, $\neg$, $\wedge$ and $\vee$ are orthogonal operators.

To compute an orthogonal operator [15] on TDGs, just follow this algorithm:

$$Op^{TDG}(k_1, \ldots, k_n) = Op(k_1, \ldots, k_n)$$
$$Op^{TDG}(f_1, \ldots, f_n) =$$

[4] See fig. 3 for an example.

```
let x = inf ⋃_{1≤i≤n} V(f_i),
let T_1 = Op^{TDG}(f_{1x̄}, ..., f_{nx̄}) and T_2 = Op^{TDG}(f_{1x}, ..., f_{nx})
if T_1 = T_2 then T_1
if the sign of T_2 is + Then (+, N(x, T_1, T_2))
if the sign of T_2 is − Then (−, N(x, ¬T_1, ¬T_2))
```

The proof of this algorithm is done by induction on $|\bigcup_{1\leq i\leq n} \mathcal{V}(f_i)|$.

If we keep in memory the intermediate results, then the total cost in time of the operator is $O(|f_1| \times \ldots \times |f_n|)$, where $|f_i|$ is the number of nodes of the TDG representing f_i. So most often[5], calculi of orthogonal operators over TDG are quite fast.

3 Abstract Interpretation

3.1 A Rapid Presentation

For generality of the results, we will use the most general framework as developped in [16].We will note the abstract semantic domain $\mathcal{P}^\sharp$ and the concrete semantic domain $\mathcal{P}^\natural$.

Following this general framework, the concrete semantics of a program is often given by the limit of the iteration of a *concrete semantic function*, $F^\natural$, starting from a basis $\perp^\natural$, and using an inductive join $\amalg^\natural$ to go to limit ordinals.

$$\begin{cases} F^{\natural 0} \stackrel{\text{def}}{=} \perp^\natural \\ F^{\natural \lambda+1} \stackrel{\text{def}}{=} F^\natural(F^{\natural\lambda}) \\ F^{\natural\lambda} \stackrel{\text{def}}{=} \amalg^\natural_{\beta<\lambda} F^{\natural\beta} \quad \text{when } \lambda > 0 \text{ is a limit ordinal} \end{cases}$$

To ensure convergence, $\mathcal{P}^\natural$ is often associated to a complete lattice structure, the limit of the iteration being then the least fixpoint of $F^\natural$. The same ideas apply to find the abstract semantics of a program.

The relation between the concrete and abstract semantic can be described by a *soundness relation* σ. $\langle c, a\rangle \in \sigma$ meaning that a is a safe approximation of the property c. Moreover, one will want the approximation not only to be safe but to be "good". To define this notion, abstract interpretation uses an *approximation order* on semantics, $\preceq$. σ is then supposed to respect the approximation order, that is to say if $a \preceq^\sharp a'$ and $\langle c, a\rangle \in \sigma$ then $\langle c, a'\rangle \in \sigma$. In the most ideal case, there will exist one best approximation for each property of $\mathcal{P}^\natural$. It will be given by an abstract function α.

Sometimes, there is no better approximation or many, or one but if used, the computation of the abstract property (possibly obtained by an abstract iteration[6]) may be too long (possibly infinite...). A solution here is the use of a

[5] see section 2.2.

[6] that is to say the limit of the $F^{\sharp\lambda}$.

widening operator. A widening operator is a partial function $\triangledown^\sharp$ from $\wp(\mathcal{P}^\sharp)$ to $\mathcal{P}^\sharp$ such that:

$$(\triangledown^\sharp A \text{ exists}) \Rightarrow \forall c \in \mathcal{P}^\natural \colon (\exists a \in A \colon \langle c, a\rangle \in \sigma) \Rightarrow (\langle c, \triangledown^\sharp A\rangle \in \sigma)$$

Then we can use the following abstract iteration with widening:

$$\begin{cases} F^{\sharp\uparrow 0} \stackrel{\text{def}}{=} \perp^\sharp \\ F^{\sharp\uparrow\lambda+1} \stackrel{\text{def}}{=} \triangledown^\sharp\{F^{\sharp\uparrow\lambda}, F^\sharp(F^{\sharp\uparrow\lambda})\} \\ F^{\sharp\uparrow\lambda} \stackrel{\text{def}}{=} \triangledown^\sharp_{\beta<\lambda} F^{\sharp\uparrow\beta} \quad \text{when } \lambda > 0 \text{ is a limit ordinal} \end{cases}$$

3.2 Using TDGs

Most abstract interpretations are designed with the idea of implementation in mind. So the elements of most abstract semantics domain are computer representable. It means that there exists an injection from $\mathcal{P}^\sharp$ to $\bigcup_n \{0,1\}^n$ which is equivalent to $\mathcal{P}^\sharp$ is countable. As $\mathcal{B}$ is countable too, it means that we can theoretically find an injection from most abstract semantics domains to $\mathcal{B}$. The problem is that there are many possible such injections, and certainly not all are interesting in the sense that they make the abstract interpretation use the properties of the TDG. For example, an injection that to each element of $\mathcal{P}^\sharp$ associates a variable (and whatever function from $\{0,1\}$ to $\{0,1\}$) will just consume more space.

There is no general method (yet) to find a good encoding relation β. It should be at least the inverse of a function, so that to different values of $\mathcal{P}^\sharp$ cannot have the same representation. The abstract function, if any, will induce an operator $F^\flat$, which should be easily computed, but we cannot ask that it is orthogonal, as the only orthogonal operators from $\mathcal{B}$ to $\mathcal{B}$ are $\mathbf{1}$, $\mathbf{0}$, the identity and the negation. Instead, it can be a polynomial of orthogonal operators.

Here we can see that there can be different possibilities even for what is encoded, as it can be convenient to code the abstract domain, or the abstract functions or both. Of course, if both can be encoded easily, it will lead to better results.

Lifting an Abstract Interpretation. Abstract interpretation of higher order functions is often performed by lifting the abstract domain of first order functions. So, it can be interesting to have a general way of coding lifted domains with TDGs. let $\mathcal{P}_1^\sharp$ and $\mathcal{P}_2^\sharp$ be two abstract semantics encoded into TDGs by β_1 and β_2. We'll suppose moreover that $\mathcal{P}_1^\natural(\preceq_1^\natural) \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} \mathcal{P}_1^\sharp(\preceq_1^\sharp)$ and $\mathcal{P}_2^\natural(\preceq_2^\natural) \underset{\gamma_2}{\overset{\alpha_2}{\rightleftarrows}} \mathcal{P}_2^\sharp(\preceq_2^\sharp)$ are Galois connections[7]. As suggested in [2], such Galois connections can be lifted to functions:

$$\mathcal{P}_1^\natural \overset{m}{\longmapsto} \mathcal{P}_2^\natural(\preceq^\natural) \underset{\lambda g.\gamma_2 \circ g \circ \alpha_1}{\overset{\lambda f.\alpha_2 \circ f \circ \gamma_1}{\rightleftarrows}} \mathcal{P}_1^\sharp \overset{m}{\longmapsto} \mathcal{P}_2^\sharp(\preceq^\sharp)$$

[7] i.e. $\forall c \in \mathcal{P}^\natural, \forall a \in \mathcal{P}^\sharp : (c \preceq^\natural \gamma(a)) \iff (\alpha(c) \preceq^\sharp a)$.

is a Galois connection too, assuming $\preceq$ is the pointwise ordering[8], and $A \overset{m}{\longmapsto} B$ is the set of monotonic functions from A to B.

So the lifted semantic domain contains functions from $\mathcal{P}_1^\sharp$ to $\mathcal{P}_2^\sharp$. It means that if we wanted to directly extend the encoding to the lifted domain, we would need functions over boolean functions, which are not directly representable by TDGs. So let's code the following set: let $B_n^\uparrow \stackrel{\text{def}}{=} (\{0,1\}^n \to \{0,1\}) \to \mathcal{B}$, $Var_n^\uparrow$ being the set of variables used in $B_n^\uparrow$. $Var^\uparrow \stackrel{\text{def}}{=} \bigcup_n Var_n^\uparrow$ and $B^\uparrow \stackrel{\text{def}}{=} \bigcup_n B_n^\uparrow$. We cannot directly translate it into TDGs because we cannot make a binary decision after testing a variable of $Var^\uparrow$. A solution then is to transform the testing of a functional variable into a sequence of binary tests. To perform that, we will need two functions sequences, $then_n$ and $else_n$ going from $Var_{n+1}^\uparrow$ to $Var_n^\uparrow$. The idea is that the value of a variable f of $Var_{n+1}^\uparrow$ applied to a boolean is either $then_n(f)$ either $else_n(f)$. To ensure that we are really making a choice, we'll add the following properties to $then_n$ and $else_n$: $\forall f, g \in Var_{n+1}^\uparrow$, $then_n(f) \neq else_n(g)$ and $then_n$ and $else_n$ are injections. Then we can define $v : Var^\uparrow \to \wp(Var)$

$$\begin{aligned} v(x) &\stackrel{\text{def}}{=} \{b(x)\} \quad \text{if } x \in Var_0^\uparrow \\ v(f) &\stackrel{\text{def}}{=} v(else_n(f)) \cup v(then_n(f)) \quad \text{if } f \in Var_{n+1}^\uparrow \end{aligned}$$

where b is a bijection from $Var_0^\uparrow$ to Var. So, $v(f)$, is a set of variables $\{f_1, \ldots, f_{2^n}\}$. From this set of variable we will need to build the associated boolean function $build : \{0,1\}^{2^n} \to \mathcal{B}$, $build\{x\} \stackrel{\text{def}}{=} x$ and $build\{x_1, \ldots, x_{2n}\} \stackrel{\text{def}}{=} \lambda x. \neg x \wedge build\{x_1, \ldots, x_n\} \vee x \wedge build\{x_{n+1}, \ldots, x_{2n}\}$. We use a λ-term here because the variable appearing after the λ will always be substituted, so the name of the variable has no importance.

So, for example, substituting the variable f of $Var_1^\uparrow$ with the function $\lambda x.\neg x$ is the same as substituting $v(f) = (f_1, f_2)$ with $(1, 0)$.

We can now code $B^\uparrow$:

$$down(\lambda f.g(x_1, \ldots, x_n)) \stackrel{\text{def}}{=} \lambda y_1, \ldots, y_m.g[f/build\{y_1, \ldots, y_m\}](x_1, \ldots, x_n)$$

where $\{y_1, \ldots, y_m\} = s^\star(v(f))$, s a permutation such that $y_m <^v x_1$. One should notice that the number of variables y is well defined, which wouldn't be the case with $(\bigcup_n (\{0,1\}^n \to \{0,1\})) \to \mathcal{B}$.

Example: $down(\lambda f.\lambda x.fx) = \lambda f_1.\lambda f_2.\lambda x.\neg x \wedge f_1 \vee x \wedge f_2$.

We now have a coding of $\mathcal{P}_{1\to 2}^\sharp$, if for all variable of $\mathcal{P}_1^\sharp$ there exist an N such as all instantiation of the variable is coded in $\mathcal{B}_n$ with $n \leq N$. Then β_1^v of such a variable is a variable in $Var_N^\uparrow$. So

$$\beta_{1\to 2}(\lambda f.g) \stackrel{\text{def}}{=} down(\lambda \beta_1^v(f).\beta_2(g))$$

This coding is interesting for abstract functions too, because if $g = \text{lfp}(F_2)$ then $\lambda f.g = \text{lfp}(F_{1\to 2})$ where $F_{1\to 2}(\lambda x.y) \stackrel{\text{def}}{=} \lambda x.F_2(y)$. So if $F_2^\sharp$ is coded into TDG, $F_{1\to 2}^\sharp$ can be coded into TDG too.

[8] $f \preceq g \iff \forall x \in \mathcal{P}_1,\ f(x) \preceq_2 g(x)$.

Moreover, this coding will even *provide* a way to code the abstract functions of $\mathcal{P}_1^\sharp$, as they are just functions from $\mathcal{P}_1^\sharp$ to $\mathcal{P}_1^\sharp$.

A Widening Operator on TDGs. The question of the size of a TDG is essential in its efficiency; not only because taking less room is efficient in itself, but as seen in section 1, the speed of operators upon it depends directly of its size. A solution to reduce the size could be to use the less powerful representation without explicit variables and try different ordering for the variable, but so far, no really satisfactory solution have been proposed, and anyway there will always be cases exponential whatever the ordering. So the solution — specific to abstract interpretation — proposed is a widening operator based on the size of the TDG. This widening operator is very general, and can be used every time the size of the abstract domain is too big: in such a case, the encoding of a single element of the abstract domain can be too much for practical manipulation. Using this widening operator, it is possible to chose an approximate solution that is compact enough to be representable on a computer.

Prerequisites and Characteristics. This widening operator is closely related to the approximation ordering upon $\mathcal{P}^\flat$, $\preceq^\flat$ induced by $\preceq^\sharp$, which should be compatible with the structure the TDGs. In fact what the widening operator exactly need is a way to compute the least upper bound of two TDGs for $\preceq^\flat$, and, as this operation will be essential to the widening operator, the cheaper the better.

Then, the widening operator requires a limit l and a boolean function f. The result $\nabla(l, f)$ is a boolean function g such that $|g| \leq l$ and $f \preceq^\flat g$. To ensure that it is always possible (for all l positive), it is suggested that $(+, 1)$ or $(-, 1)$ represent the top of $\mathcal{P}^\sharp$.

This operator can be used to produce a very classical widening operator as defined in the beginning of this section: $\nabla^\flat A \stackrel{\text{def}}{=} \nabla(l(\max(A)), \max(A))$ where $\max(A)$ is, if it exists, the maximum of A for the computational ordering[9] ($\sqsubseteq^\flat$), and l a function that gives the limit.

But this operator can also be used in a way closer to static widening, if the abstract functions are coded into TDG too, because if the TDG used to represent the abstract function is too big, each step of the iteration will be too long, and sometime, the size of the TDG representing the iterates will be directly related to the size of the TDG representing the abstract function. So it can be very profitable to approximate directly the abstract function, justified by the following property:

Property 1 *F_1 and F_2 monotonic functions (for $\sqsubseteq$). If $\forall f\ F_1(f) \preceq F_2(f)$ and F_1 or F_2 are monotonic for $\preceq$ then*

$$\mathit{lfp}(F_1) \preceq \mathit{lfp}(F_2)$$

Proof. $f \preceq g$ implies $F_1(f) \preceq F_2(g)$ because $F_1(f) \preceq F_1(g)$ by monotonicity and $F_1(g) \preceq F_2(g)$ by hypothesis. $F_1(\bot) \preceq F_2(\bot)$ by hypothesis. The property comes by induction on the iterates. □

[9] the ordering used to ensure termination of the iterations.

Algorithm. The principle problem for this widening operator is to find the best possible g such that $|g| \leq l$, in a reasonable amount of time. Searching the best solution[10] is *not* reasonable as it would theoretically require to explore all the possible derivations of a given TDG, which is exponential.

The idea here is to force the apparition of one of the reduction steps described in section 1. To obtain **sharing**, we just take two nodes of the TDG and, to make them equal, replace them by the least upper bound of the two nodes. To obtain **elimination of superfluous nodes**, we replace a node $N(x, T_1, T_2)$ by the least upper bound of T_1 and T_2.

The algorithm proceeds by steps: each step, if the size of the TDG is above the limit, then try each of the reductions above and take the better one; repeat. The better one is the one with the higher rate (number of nodes above the limit gained)/(cost of the reduction), where the cost of the reduction is if the reduction is a sharing of T_1 and T_2 then $\mathrm{cost}(T_1 \to T') \times \mathrm{mult}(T_1) + \mathrm{cost}(T_2 \to T') \times \mathrm{mult}(T_2)$, and if it is the elimination of T then $(\mathrm{cost}(T_1 \to T') + \mathrm{cost}(T_2 \to T')) \times \mathrm{mult}(T)$. Each reduction implies taking the least upper bound T' of two TDGs T_1 and T_2. The computation of the least upper bound is supposed to give $\mathrm{cost}(T_1 \to T')$ and $\mathrm{cost}(T_2 \to T')$ [11]. mult is the multiplicity of the node, that is to say the number of time the node would appear in the Shannon's tree representation of the TDG, so that changing a node shared by many others would cost more than changing one only used by one other node.

Complexity. Each reduction forced will not automatically reduce the size of the TDG, as the least upper bound may contain more new nodes than gained through the reduction. However, as there will always, for a TDG greater than 1, be a node $N(x, (-,1), (+,1))$[12], if $(+,1)$ or $(-,1)$ represents the top of $\mathcal{P}^\sharp$, there will always be a reduction that reduces the size of the TDG. So if the limit is positive, each step the size of the TDG will either decrease or be less than or equal to the limit. It means that the number of steps is less than the difference between the limit and the size of the TDG, which is still too much, as this difference may be exponential.

To come faster to a size closer to the limit, we will use a less refined algorithm, that assumes that $(+,1)$ or $(-,1)$ represents the top: for each node such that the size of the TDG without that node [13] is between the limit and the limit plus half the difference between the limit and the size of the TDG, we try to replace them by the top and take the one that gives the best result. That way, each step of this algorithm will at least half the difference between the limit and the size of the TDG.

The more precise algorithm requires that each pair of node be tested. The multiplicity of each node can be calculated in a time polynomial to the number of

[10] That is to say the min (for $\preceq^\flat$) of all the possible solutions.

[11] This cost is supposed to express the loss of precision; for example it could be the length of the maximum chain between T_i and T'.

[12] When x is the last (for $<^v$) of the variables appearing in the TDG, it is the only possible node on x.

[13] i.e. after replacing this node by the top.

nodes, by going through the TDG and tagging the nodes. So if the computation of the least upper bound (plus the computation of the costs) is polynomial in the size of the TDGs, then the more precise algorithm is polynomial in the size of the TDG.

So combining the two algorithms, such that if the difference of the size of the TDG and the limit l is bigger than $P(l)$ where P is a polynomial, then use the rough algorithm else the other one, and assuming that the limit is polynomial in the number of variables of the TDG, then the global algorithm is polynomial in the number of variables.

4 A Complete Example: Strictness Analysis

Definition 3. A function f is said to be *strict* in one of its arguments x if whenever the evaluation of that argument fails, the evaluation of f fails

The evaluation fails if it ends with an error or if it does not terminate.

The goal of a strictness analysis is to determine whether a function is strict in any of its arguments. This can be useful for example in the optimization of a call-by-need compiler: the principle of such a compiler is to keep the arguments of functions in a closure until it is first needed in the evaluation of the function and then evaluate it. If a function is strict in an argument, then that argument will be needed anyway, so the compiler can evaluate the argument anyway[14], saving the room for the closure.

Strictness analysis is a good example of application of the use of TDGs because it is an analysis that can be useful, in a compiler for example, but the more precise abstract interpretations known so far are too slow to be used at higher order.

4.1 Standard Strictness Analysis

What we call standard analysis is the abstract interpretation that will be coded into TDGs. The analysis we will use as a basis is the well-known one developed by Alan Mycroft, that still seems to be one of the more precise, and that have the advantage of being already coded into boolean functions.

The Concrete Domain. Mycroft's analysis concerns first order functions from base types to base types. The concrete semantic domain $\mathcal{P}^\natural$ is the set of relations from $\mathcal{D}$ to $\mathcal{D}$ where $\mathcal{D}$ is a complete domain with infimum $\bot$ and the values from the base types, such as integers. $\bot^\natural \stackrel{\text{def}}{=} \lambda x.\bot$. The concrete semantic function is constructed by induction on the syntax of the expression defining the function: $F^\natural = S[\![\texttt{f(x)=}e]\!]$.

$$S[\![\texttt{f(x)=b(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}]\!](g) \stackrel{\text{def}}{=} \{(x, \underline{b}(v_1, \ldots, v_n)) | \bigwedge_{1 \le i \le n} (x, v_i) \in S[\![\texttt{f(x)=}e_i]\!](g)\}$$

[14] Assuming there is no side effect.

$$S[\![\texttt{f(x)=x}]\!](g) \stackrel{\text{def}}{=} \{(x,x) | x \in \mathcal{D}\}$$

$$S[\![\texttt{f(x)=f(}e\texttt{)}]\!](g) \stackrel{\text{def}}{=} \{(x,w) | (x,v) \in S[\![\texttt{f(x)=}e]\!](g) \wedge (v,w) \in g\}$$

Where `b` are constants of the language, such as $+$, integers or the conditional. $\underline{b}$ is the corresponding constant on $\mathcal{P}^\natural$. For example $\underline{2} = 2$, and $\underline{ite}(x_1, x_2, x_3) =$ if $x_1 = \perp$ then $\perp$, if $x_1 =$`true` then x_2 and if $x_1 =$`false` then x_3.

The Abstract Domain. The abstract domain introduced by Mycroft is the set of monotonic functions from $\{0,1\}$ to $\{0,1\}$, with the ordering $0 \preceq^\sharp 1$[15], which can be interpreted as $\lambda x.0$ the function never terminates, $\lambda x.x$ the function is strict in x, and $\lambda x.1$ we do not know. $\perp^\sharp \stackrel{\text{def}}{=} \lambda x.0$. The abstract semantic function is defined by induction on the syntax too:

$$S^\sharp[\![\texttt{f(x)=b(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}]\!](g^\sharp) \stackrel{\text{def}}{=} b^\sharp(S^\sharp[\![\texttt{f(x)=}e_1]\!](g^\sharp), \ldots, S^\sharp[\![\texttt{f(x)=}e_n]\!](g^\sharp))$$

$$S^\sharp[\![\texttt{f(x)=x}]\!](g^\sharp) \stackrel{\text{def}}{=} \lambda x.x$$

$$S^\sharp[\![\texttt{f(x)=f(}e\texttt{)}]\!](g^\sharp) \stackrel{\text{def}}{=} g^\sharp \circ S^\sharp[\![\texttt{f(x)=}e]\!](g^\sharp)$$

$b^\sharp$ represents `b` on $\mathcal{P}^\sharp$. For example $2^\sharp = 1$ and $ite^\sharp(f_1, f_2, f_3) = f_1 \wedge (f_2 \vee f_3)$.

The Relations Between the Two Semantics. The soundness relation between $\mathcal{P}^\natural$ and $\mathcal{P}^\sharp$ is described by a Galois connection, $\mathcal{P}^\natural \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} \mathcal{P}^\sharp$:

$$\alpha(f)(0) \stackrel{\text{def}}{=} \text{if } \{x | (\perp, x) \in f\} = \{\perp\} \text{ then } 0 \text{ else } 1$$

$$\alpha(f)(1) \stackrel{\text{def}}{=} \text{if } \{y | x \in \mathcal{D} \wedge (x,y) \in f\} = \{\perp\} \text{ then } 0 \text{ else } 1$$

$$\gamma(\lambda x.0) \stackrel{\text{def}}{=} \lambda x.\perp$$

$$\gamma(\lambda x.x) \stackrel{\text{def}}{=} \{(\perp,\perp)\} \cup ((\mathcal{D} - \{\perp\}) \times \mathcal{D})$$

$$\gamma(\lambda x.1) \stackrel{\text{def}}{=} \mathcal{D} \times \mathcal{D}$$

To ensure that $F^\sharp$ is a good approximation of F we shall make a few more assumptions on the constants:

$$\text{if } \forall i,\ \alpha(f_i) \preceq^\sharp g_i^\sharp \text{ then}$$

$$b^\sharp(g_1^\sharp, \ldots, g_n^\sharp) \succeq^\sharp \alpha(\{(x, \underline{b}(v_1, \ldots, v_n)) | \bigwedge_{1 \le i \le n} (x, v_i) \in f_i\})$$

Then

Property 2 $\alpha(\mathit{lfp}(F)) \preceq^\sharp \mathit{lfp}(F^\sharp)$.

[15] The computational ordering is the same as the approximation ordering.

Proof. By induction on the syntax, we shall first prove that $\alpha(f) \preceq^\sharp g^\sharp$ implies that $\alpha(F(f)) \preceq^\sharp F^\sharp(g^\sharp)$, then as $\alpha(\bot^\natural) = \bot^\sharp$ the inequation on the fixpoints will come by induction on the iterates.

So let's suppose $\alpha(f) \preceq^\sharp g^\sharp$.

$$\begin{aligned}
&\alpha(S[\![\mathtt{f(x)=b(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)}]\!](f)) \\
&\quad = \alpha(\{(x, \underline{b}(v_1, \ldots, v_n)) | \bigwedge_{1\le i\le n} (x, v_i) \in S[\![\mathtt{f(x)=}e_i]\!](f)\}) \\
&\quad \preceq^\sharp b^\sharp(S^\sharp[\![\mathtt{f(x)=}e_1]\!](g^\sharp), \ldots, S^\sharp[\![\mathtt{f(x)=}e_n]\!](g^\sharp)) \\
&\quad \preceq^\sharp S^\sharp[\![\mathtt{f(x)=b(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)}]\!](g^\sharp)
\end{aligned}$$

The first line is by definition of S. Then by hypothesis of induction and then the property of the abstract constants. Then by definition of $S^\sharp$.

$$\begin{aligned}
\alpha(S[\![\mathtt{f(x)=x}]\!](f)) &= \alpha(\{(x, x) | x \in \mathcal{D}\}) \\
&= \lambda x.x \\
&= S^\sharp[\![\mathtt{f(x)=x}]\!](g^\sharp)
\end{aligned}$$

For the last step of the proof we need a few more results on the composition of relations. $R_1 \circ R_2 \stackrel{\text{def}}{=} \{(x, w) | (x, v) \in R_2 \wedge (v, w) \in R_1\}$. Suppose $\alpha(R_1) \circ \alpha(R_2)(a) = 0$. If $\alpha(R_2)(a) = 0$ then $\{y | x \in A \wedge (x, y) \in R_2\} = \{\bot\}$[16] and $\{y | (\bot, y) \in R_1\} = \{\bot\}$, so $\{y | x \in A \wedge (x, v) \in R_2 \wedge (v, y) \in R_1\}$ is $\{\bot\}$, so $\alpha(R_1 \circ R_2)(x) = 0$. If $\alpha(R_2)(a) = 1$ then $\{y | (x, y) \in R_1\} = \{\bot\}$ so $\{y | x \in A \wedge (x, v) \in R_2 \wedge (v, y) \in R_1\}$ is $\{\bot\}$, so $\alpha(R_1 \circ R_2)(x) = 0$. It means that $\forall R_i$, $\alpha(R_1 \circ R_2) \preceq^\sharp \alpha(R_1) \circ \alpha(R_2)$.

$$\begin{aligned}
\alpha(S[\![\mathtt{f(x)=f(}e\mathtt{)}]\!](f)) &= \alpha(\{(x, w) | (x, v) \in S[\![\mathtt{f(x)=}e]\!](f) \wedge (v, w) \in f\}) \\
&= \alpha(f \circ S[\![\mathtt{f(x)=}e]\!](f)) \\
&\preceq^\sharp \alpha(f) \circ \alpha(S[\![\mathtt{f(x)=}e]\!](f)) \\
&\preceq^\sharp g^\sharp \circ S^\sharp[\![\mathtt{f(x)=}e]\!](g^\sharp) \\
&\preceq^\sharp S^\sharp[\![\mathtt{f(x)=f(}e\mathtt{)}]\!](g^\sharp)
\end{aligned}$$

The first line is the definition of S. Then the definition of the composition of relations. Then what was just proved above on composition and α. The last lines use the fact that $\alpha(f) \preceq^\sharp g^\sharp$ by hypothesis, $\alpha(S[\![\mathtt{f(x)=}e]\!](f)) \preceq^\sharp S^\sharp[\![\mathtt{f(x)=}e]\!](g^\sharp)$ by hypothesis of induction, and $g^\sharp$ is monotonic as every function in $\mathcal{P}^\sharp$. $\square$

It is interesting to notice that Mycroft's analysis gives more than just the strictness result: it gives results that can be useful in further analysis using this function. For example `f(x)=f(x)` will give $\lambda x.0$ so f is strict in x. With the only information that f is strict in x we cannot say that g defined by `g(y)=f(0)` is strict too.

[16] If $a = 0$ then $A = \{\bot\}$ and if $a = 1$ then $A = \mathcal{D}$.

4.2 The Encoding

To code the abstract domain, we will merely add variable names and $\mathcal{P}^\flat$ will be $\mathcal{B}_1$. Abstract functions could be coded using the method presented in the previous section, as they are functions from $\mathcal{B}_1$ to $\mathcal{B}_1$. The problem, when dealing with the higher orders, is that, the size of the type being increasing, and as each step of the iteration we would require every possible value of the previous iterate, we would loose all the interest of the TDG for recursive functions. We will prefer to code each recursive call by a new variable, keeping the arguments of the recursive call. That way, each step of the iteration we only need to make substitutions in the previous iterate, the number of which will be polynomial in the size of the program.

So this abstract interpretation can be easily lifted to higher order functions. As the encoding is very close to the abstract domain, we can have a better *build* function that associates the boolean function to the set of variables, keeping only monotonic functions: $build\{x_1, \ldots, x_{2n}\} \stackrel{\text{def}}{=} \lambda x.build\{x_1, \ldots, x_n\} \vee x \wedge build\{x_{n+1}, \ldots, x_{2n}\}$.

Given $\mathcal{P}^\flat$ for higher order functions, here is the abstract function:

$$\begin{aligned}
S^\flat[\![\mathtt{b(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)}]\!]\rho(g^\flat) &\stackrel{\text{def}}{=} b^\flat(S^\flat[\![e_1]\!]\rho(g^\flat), \ldots, S^\flat[\![e_n]\!]\rho(g^\flat)) \\
S^\flat[\![\mathtt{x}]\!]\rho(g^\flat) &\stackrel{\text{def}}{=} \rho(\mathtt{x}) \\
S^\flat[\![e_1 e_2]\!]\rho(g^\flat) &\stackrel{\text{def}}{=} S^\flat[\![e_1]\!]\rho(g^\flat) S^\flat[\![e_2]\!]\rho(g^\flat) \\
S^\flat[\![\lambda\mathtt{x}.e]\!]\rho(g^\flat) &\stackrel{\text{def}}{=} S^\flat[\![e]\!]\rho[x \rightarrow newvar](g^\flat)
\end{aligned}$$

ρ is an environment function. It maps variables (strings) to TDGs. If the variable is associated to a previously analyzed function, it gives the TDG representing the result. If it is a free variable, it gives the TDG as constructed in the previous section representing a variable function. This TDG is constructed by *newvar* which needs to know of course the type of the variable. If the variable represents the function defined (recursive call), then ρ returns a single boolean variable, and each time it is applied, it is replaced by a new variable that will represent the application.

Example:

`s x y z = (x z)(y z).`

$\rho(\mathtt{x}) = \lambda a.\lambda b.x_1 \vee x_3 \wedge a \vee (x_2 \vee x_4 \wedge a) \wedge b$.

$\text{lfp}(S^\flat[\![\mathtt{(x\ z)(y\ z)}]\!]\rho)$ is the TDG represented on fig. 4. As, if $(x_1, x_2, x_3, x_4) = (0, 0, 0, 0)$, the TDG is 0, s is strict in x. But if $x_1 = 1$, the TDG is 1, so the interpretation tells nothing about the strictness of s in y or z.

The ordering on $\mathcal{P}^\flat$ is the implication, so the max of two TDG is very easy to compute, it is $\wedge$, which is orthogonal. So we can use the widening operator on that example. Moreover, the pointwise ordering on the abstract functions leads to the same ordering (implication) on the representations of the abstract function, so that the same widening can be applied to the abstract set and to the abstract functions.

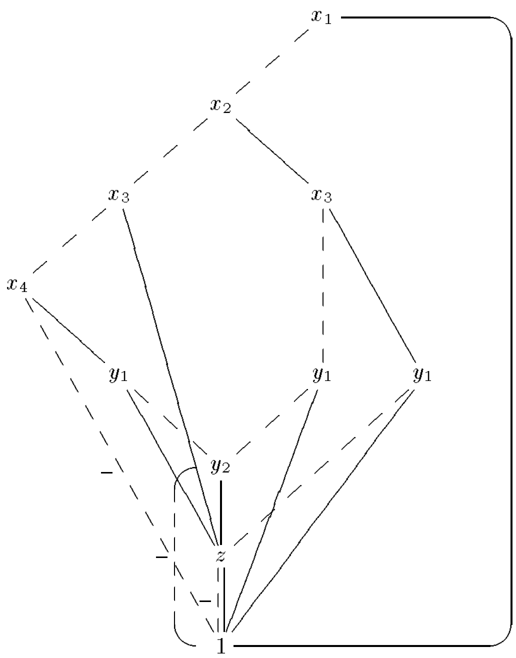

Fig. 4. `s x y z = (x z)(y z)`

4.3 Practical Results

Strictness analysis have been implemented using TDGs[17], and tested on examples given by Sebastian Hunt to compare the efficiency of this implementation with the one he developed based on 'frontiers'. All the results given bellow are for the interpreted version (in camllight) and could be fasten by compilation, and the implementation of Sebastian Hunt was only a prototype implementation, so the comparison might be unfair.

n^o	frontiers	TDG
1	10 sec	3 sec
2	5 min	2 sec
3	30 min	4 sec
4	never ended	1 hour

The first example is a quite classic nqueen solver, using few higher order functions. The results are quite good with both methods. The second one uses map and foldr[18]. The third one uses foldr at a higher order, applying it to append, so the result of the analysis is much bigger.

[17] The TDG package used for this implementation is the one developed by Brace, Rudell and Bryant, as a subset of COSMOS.

[18] foldr is the classical function that applies recursively a binary function to a list.

The fourth example analyzes foldr written in continuation passing style, leading to a dramatic increase in the type order. Two functions are analyzed with type (α `list` $\rightarrow$ α `list` $\rightarrow$ (α `list` $\rightarrow \beta$) $\rightarrow \beta$) $\rightarrow$ α `list list` $\rightarrow$ α `list` $\rightarrow$ (α `list` $\rightarrow \beta$) $\rightarrow \beta$. It is interesting because the result is so huge that it cannot be computed and intermediate results couldn't be stored by the computer. So it seems to be an example where the TDG representation is exponential, that shows the usefulness of the widening presented above. Of course, the result of the analyze is approximate due to the use of this operator.

For the last example, a good alternative to a complete analysis was presented in [20], that gives results in a few seconds, but it cannot be used for separate compilation.

5 Conclusion

This approach proved to be efficient in strictness analysis, and could be advantageously used in many other abstract interpretation. But the last example shows that it may still be too slow to be usable in practice. This work is totally compatible with the theoretical framework of abstract interpretation, so it could be used with other works on this subject. The idea of lazy evaluation of abstract functions from Ferguson and Hugues was mentioned above, but the results of Baraki on interpretation of polymorphic functions in [19] would be very useful for this approach too, as it could lead to a compact analysis usable in separate analysis. The author believes that the combination of these techniques could give analyzers based on abstract interpretation for higher order functions efficient enough to be usable in practice.

Acknowledgements

I would particularly like to thank Patrick Cousot, for his very heplful advises and support, and for he had the good idea of using this representation. All the members of his team helped me too for every little problem. Thanks to Sebastian Hunt too for his very useful examples.

References

1. C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions AIEE*, **57**:305–316. 1938.
2. P. Cousot & R. Cousot. Static determination of dynamic properties of recursive procedures. *IFIP Conference on Formal Description of Programming Concepts, St-Adrews, N. B., Canada*, pp. 237–277. 1977.
3. S. B. Akers. Binary decision diagrams. *IEEE Transactions on computers*. 1978.
4. P. Cousot & R. Cousot. Constructive version of Tarki's fixed point theorems. *Pacific Journal of Mathematics*. 1979.
5. R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. Comput. C-35*, pp. 677–691. 1986.

6. G. L. Burn, C. Hankin & S. Abramsky. Strictness analysis for higher-order functions. *Science of computer programming 7*, pp. 249–278. 1986.
7. P. Hudak & J. Young. Higher order strictness analysis in untyped lambda calculus. *ACM*. 1986.
8. J. P. Billon. Perfect normal forms for discrete programs. *Technical report 87039 BULL*. 1987.
9. J. C. Madre & J. P. Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. *Proc. of the 25th DAC*. 1988
10. A. R. Brayton, B. Lin & H. J. Touati. Don't care minimization of multi-level sequential logic network. *Proc. of ICCAD'90*. 1990.
11. J. C. Madre, C. Berthet & O. Coudert. New ideas in symbolic manipulation of finite state machines. *Proc. of ICCAD'90*. 1990.
12. J. Schwable & K. L. McMillan. Formal verification of the encore gigamax cache. *International Symposium on Shared Memory Multiprocessor*. 1991.
13. D. Taubner, E. Enders & T. Filkorn. Generating BDDs for symbolic model checking in ccs. *Proc. of CAV'91*, pp. 203–213. 1991.
14. H. J. Touati, H. Savoj & R. K. Brayton. Extracting local don't care for network optimization. *Proc. of ICCAD'91*. 1991.
15. R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys, Vol. 24*. pp. 293–318. 1992.
16. P. Cousot & R. Cousot. Abstract interpretation framework. *Journal of logic and computation*, pp. 511–547. 1992.
17. Corsini, Musumbu, Rauzy & Le Charlier Efficient Bottom-up Abstract Interpretation of Logic Programs by means of Constraint Solving. *PLILP '93*. 1993.
18. C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE. *These de l'université de Grenoble 1*, chap 11. 1992.
19. G. Baraki. Abstract Interpretation of Polymorphic Higher-Order Functions. *Computing Science research report of the University of Glasgow*. 1993.
20. A. Ferguson & J. Hughes. Fast Abstract Interpretation Using Sequential Algorithms. *Proc. of WSA '93*, pp. 45–59. 1993.