Abstract Interpretation Using Typed Decision Graphs

Laurent Mauborgne[⋆]

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

Abstract. This article presents a way of implementing abstract interpretations that can be very efficient. The improvement lies in the use of a symbolic representation of boolean functions called Typed Decision Graphs (TDGs), a refinement of Binary Decision Diagrams. A general procedure for using this representation in abstract interpretation is given; we examine in particular 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 in efficiency.

1 Introduction

One of the basic problems of program analysis is that, even theoretically speaking, there are properties of programs which cannot always be computed, such as termination. A way to circumvent this difficulty is to allow for partial or approximate answers. Abstract interpretation is the theoretical framework to design automatic program analysis based on sound approximations. Although this theory deals very well with many problems of program analysis, it may become unusable in practice when the analysis is too precise, because of the amount of memory, or time required. The goal of this article is to show that it is sometimes possible, using compact representations of boolean functions, not only to increase significantly the efficiency of the analysis, but also to balance the trade off between precision and efficiency during the analysis.

In the second section of this paper, we will describe the symbolic representation of boolean functions. In section 3, we will show how to use it in abstract interpretation. We will expose in detail the coding of higher order functions through TDGs, and the use of those graphs in conjunction with data approximation. The last section is dedicated to a complete example of abstract interpretation using TDGs: strictness analysis.

Because the most general framework of abstract interpretation is mathematical, most elements of this paper have been described mathematically. Consequently some of the principles may come through unclear. The reader who is not familiar with some concepts or does not want to read mathematical formulas

[⋆] email: Laurent.Mauborgne@ens.fr

should read the informal descriptions, which will give an idea of what is going on. On the other hand, if the reader is already familiar with one notions, he is invited to skip the informal presentation corresponding to this notion.

2 Typed Decision Graphs

Typed Decision Graphs [9], or TDGs, are powerful symbolic representations of boolean functions. They are a refinement of the well-known Binary Decision Diagrams [6], or BDD, which are already widely used in many fields, such as circuits synthesis and verification [11, 12, 15], or protocols verification [13, 14] but mostly unused in abstract interpretation (but see [20, 21]). The purpose of this paper is to show that this representation of boolean functions can in some cases have major applications in abstract interpretation.

2.1 Informal Presentation of Binary Decision Diagrams

A BDD, as introduced by Bryant in [6], is a compact representation of the Shannon tree of a boolean expression.

Shannon Trees Shannon trees are used to represent boolean expressions. They describe a way to evaluate the expression. First evaluate the value of one of the boolean variables of the expression. If this variable is `true`, then we can represent a boolean expression containing less variables, and if it is `false`, we represent another boolean expression containing less variables. If, in the end, the boolean expression does not contain any more variable, then its value is either `true` or `false`.

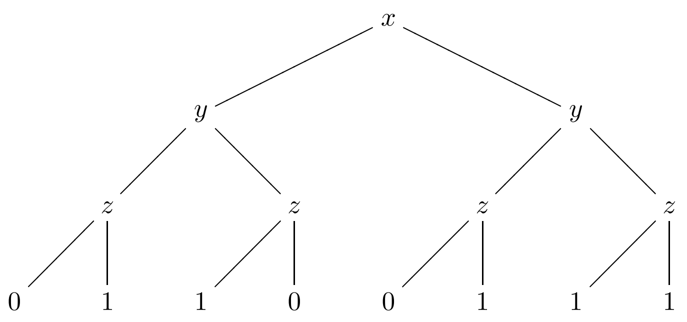
As a result, each node of a Shannon tree is associated to a variable, the left subtree represents the boolean expression when this variable is `false` and the right subtree when it's `true`.

In order to have a unique representation of a given boolean expression, the variables of the expression are to be taken in a predetermined order.

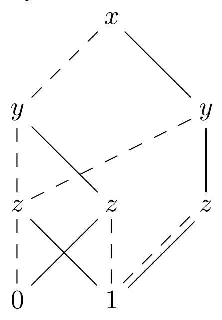
For example, let's consider the following expression: $(x\wedge y)\vee(y\wedge\neg z)\vee(z\wedge\neg y)$. We can represent this expression f using a table:

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

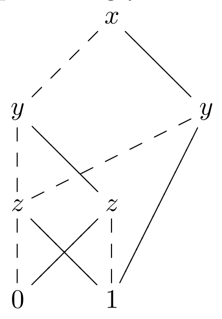
If $x < y < z$, the Shannon tree representing f will be:



Reduction Rules Once a boolean expression is represented by a Shannon tree, it is easy to see how to gain space. First, there's no need to duplicate subtrees. The action of merging redundant subtrees is called *sharing*. Instead of having a binary tree, we will have a directed acyclic graph. In order to recognize left subtrees from right subtrees we will draw the formers with dashed line. In our example, f will be represented by:



The second reduction rule is the *elimination* of useless nodes, namely nodes where the different possible values of the variable lead to the same result. After this step, we have the BDD representing f:



2.2 Formalization Work on Binary Decision Diagrams

Abstract interpretation is a theoretical and formalized approach of program analysis. So, to use BDDs in abstract interpretation we need to formalize them very precisely. We shall first define the objects encoded by BDDs, which are boolean functions and the names of the variables used to calculate them.

Let *Var* be a totally ordered set of variables. The order on *Var* will be noted $<^v$.

$Var_n \overset{\text{def}}{=} \{V \subseteq Var \mid \; |V| = n\}$, where $|V|$ is the size of the set V.

To simplify our notations, we always order the indexes of set of variables according to the order on *Var*. So when we write $\{x_1, \ldots, x_n\} \in Var_n$ it means $\forall i,\ 1 \leq i \leq n,\ x_i \in Var$ and $x_1 <^v \ldots <^v x_n$.

$$\mathcal{B}_n \overset{\text{def}}{=} Var_n \times (\{0,1\}^n \to \{0,1\})$$
$$\mathcal{B} \overset{\text{def}}{=} \bigcup_n \mathcal{B}_n$$

The pair $(\{x_1, \ldots, x_n\}, f) \in \mathcal{B}_n$, also noted $f(x_1, \ldots, x_n)$ in this paper, is the semantics of a boolean expression with n (free) variables $x_1 <^v \ldots <^v x_n$ whose value is given by the function f. The variable x alone stands for $(\{x\}, Id)$. The symbols $\neg$, $\wedge$ and $\vee$ have the usual meaning of the boolean operators "not", "and" and "or". We define $\mathcal{V}(f(x_1, \ldots, x_n)) \overset{\text{def}}{=} \{x_1, \ldots, x_n\}$.

BDDs are based on Shannon trees, whose uniqueness is insured by Shannon's expansion theorem [1]. Written 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 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 i.e. the domain is not empty, and if a word uv is in its domain, then u is in its domain too[1]. The Shannon tree representing $f(x_1, \ldots, x_n)$ is defined as follows:

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

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

As explained in the informal presentation, BDDs are compact representations of Shannon trees, obtained by enforcing the two simple reduction rules: sharing and elimination.

Sharing. This operation transforms the tree into a directed acyclic graph (DAG) by sharing isomorphic subtrees. A binary decision DAG (BDD) can be defined recursively as being either a node N of $Var \times bdd \times bdd$ or a leaf in $\{0,1\}$.

[1] uv is the concatenation of u and v

As the transformation is described by the *share* function, it is obviously still unique.

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

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 $dom(T\backslash u) \overset{\text{def}}{=} \{v | uv \in dom(T)\}$ and such that $T\backslash u(v) \overset{\text{def}}{=} T(uv)$.

The sharing results from the fact that if two subtrees are isomorphic the mathematical objects representing these subtrees are equal. The results of *share* on them are obviously identical.

Elimination of Superfluous Nodes. Once again, the transformation can be written as transformation rules; the representation is still unique.

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

After applying this rule, a BDD does no longer represent one function of $\mathcal{B}$, but all the functions whose results are the same regardless of the assignment of additional variables absent in the BDD. For example, if $\forall x, y, z,\ f(x, y, z) = g(y)$ then $f(x, y, z)$ and $g(y)$ are represented by the same BDD. This drawback does not really matter for this work, because what we really manipulate are functions from $\{0, 1\}^\omega$ to $\{0, 1\}$.

2.3 TDGs

To reduce the size of the graph even further, we go back to Shannon trees and try to produce new isomorphic subtrees. Then we will apply the same reduction rules.

Typed Shannon Trees. The idea of typed Shannon 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 as far as Shannon trees are concerned, f and $\neg f$ are identical except for the leaves: 0 becoming 1 and 1 becoming 0. So instead of having two different trees, we only need one tree and a sign. Typed Shannon trees are merely 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 $\neg f$ can be represented by T if you change $T(\epsilon)$ in $(-s, l)$.

Now, the problem is that when using simple Shannon trees and just adding signs, canonicity is lost: 0 can be represented by $(+, 0)$ or $(-, 1)$ for example. Let us simply make a choice, once for all. Here is one that provides good results for the size of the graph [10]:

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

The resulting tree is represented in fig. 1. The signs have been put on the edges instead of the labels, and only minus have been represented to get a more compact representation.

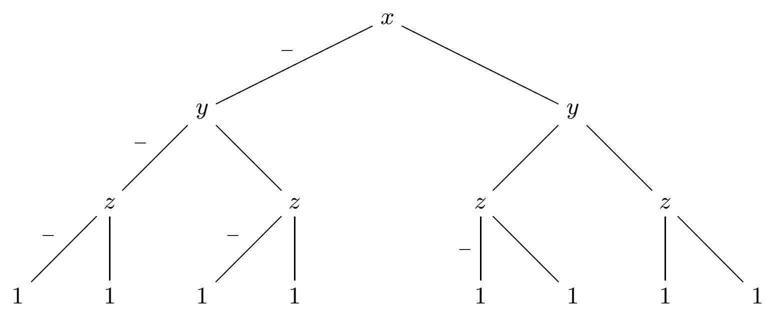


Fig. 1. Typed Shannon tree.

Resulting Graph. Now, if we simply apply the same reduction rules as for a BDD, still assuring the uniqueness of the representation, and we get Typed Decision Graphs. To know 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.

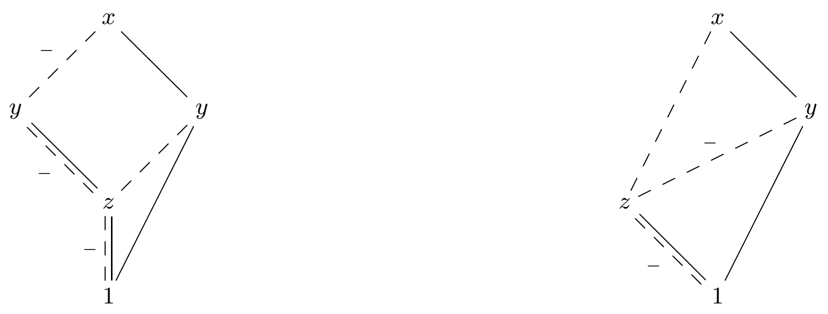


Fig. 2. The TDG for $f(x, y, z) = (x \wedge y) \vee (y \wedge \neg z) \vee (z \wedge \neg y)$, and $f(y, x, z) = (y \wedge x) \vee (x \wedge \neg z) \vee (z \wedge \neg x)$.

The size of the TDGs looks quite reasonable, and it is in most case. But there are still cases where it is exponential in the number of its variables [16]. If we restrict the representation to boolean functions without explicit variables[2],

[2] It is possible to represent boolean functions with explicit variables using boolean functions without explicit variables, so it could still be useful.

then it is sometimes possible to reduce the size of the TDG representing the function by changing the order of the variables[3]. But there are cases where the representation is still exponential, whatever the order of the variables.

2.4 Operators on TDGs

Not only does this 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 allows for a fast computation of operators is orthogonality [16].

Definition 1. *Let Op be a n-operator. Op is said orthogonal iff:*

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

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

An orthogonal operator on TDGs can be calculated by the following algorithm:

```
$Op^{TDG}(k_1, \ldots, k_n) = Op(k_1, \ldots, k_n)$
$Op^{TDG}(f_1, \ldots, f_n) =$
  let $x = \inf \bigcup_{1 \leq i \leq n} \mathcal{V}(f_i)$,
  let $T_1 = Op^{TDG}(f_{1\overline{x}}, \ldots, f_{n\overline{x}})$ and $T_2 = Op^{TDG}(f_{1x}, \ldots, f_{nx})$
  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 $(-, N(x, \neg T_1, \neg T_2))$
```

The proof of this algorithm is 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 of the time[4], calculation with orthogonal operators over TDG are quite fast.

3 Abstract Interpretation

3.1 Informal Presentation of Abstract Interpretation

Abstract interpretation is a very general and formalized framework allowing to deal with approximations. The rule of signs (positive multiplied by positive is

[3] See fig. 2 for an example.

[4] see section 2.3.

positive, etc.) can be seen as an abstract interpretation: the concrete domain (real numbers) is abstracted by approximate values in an abstract domain ({positive numbers, negative numbers, zero}), and the concrete operation (multiplication) is approximated by an abstract operation (the rule of signs).

The aspects of abstract interpretation that we will use are:

- The possibility to lift automatically an abstract interpretation. That is to say, given domains and their approximations, the possibility to approximate functions over those domains.
- Widening operators. When the semantics of a program can be expressed as the limit of the iteration of a given function (often given by the syntax of the program), the abstract semantic can also be expressed as the limit of the iteration of an abstract function. But in some cases, more approximation is needed. Then abstract interpretation provides the possibility of using a widening, which is an operator that alters the iteration, generally speeding it, but at the cost of wider approximation.

3.2 Recall of Important Aspects

Taking the most general framework [17], all the possible behaviors of programs are described in a *standard semantics*. From the point of view of abstract interpretation however, only a certain class of these behaviors is interesting. This class is the *collecting semantics*. Then the *abstract semantics* is usually an approximation of the collecting semantics[5] that keeps for example only invariance properties. All those properties are taken from sets called *semantic domains*, and one of the most important tasks of an abstract interpretation is to describe the relations between the abstract semantic domain $\mathcal{P}^\sharp$ and the concrete semantic domain $\mathcal{P}^\natural$.

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 $\bot^\natural$, and using an inductive join $\amalg^\natural$ to go to limit ordinals.

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

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 $F^\natural$ ($\text{lfp}(F^\natural)$). The same ideas apply to determine 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 sound approximation of the property c. Moreover, one will want the approximation both sound and

[5] the abstract semantics can be an approximation of whatever semantics, even another abstract semantics, so for the purpose of relations between semantics, the approximated one will always be called *concrete semantics*.

"good". To define this notion, abstract interpretation uses an *approximation order* on properties, $\preceq$. The soundness relation σ is then supposed to respect the approximation order, namely if $a \preceq^\sharp a'$ and $\langle c, a\rangle \in \sigma$ then $\langle c, a'\rangle \in \sigma$. In this case, we say that a is a better approximation than a'. 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 none or many best approximation. Even when there is only one, the computation of the abstract property (possibly obtained by an abstract iteration[6]) may be too long or even infinite. A solution for all these problems is the use of a *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:

$$\begin{cases} F^{\sharp\uparrow 0} \overset{\text{def}}{=} \bot^\sharp \\ F^{\sharp\uparrow\lambda+1} \overset{\text{def}}{=} \triangledown^\sharp\{F^{\sharp\uparrow\lambda}, F^\sharp(F^{\sharp\uparrow\lambda})\} \\ F^{\sharp\uparrow\lambda} \overset{\text{def}}{=} \triangledown^\sharp(\bigcup\{F^{\sharp\uparrow\beta} | \beta < \lambda\}) \;\; \text{when } \lambda > 0 \text{ is a limit ordinal} \end{cases}$$

If moreover there is an abstract function α, and $\triangledown^\sharp$ satisfy:

$$\triangledown^\sharp A \text{ exists} \wedge c \in \mathcal{P}^\natural \wedge a \in A \wedge \alpha(c) \preceq^\sharp a \Rightarrow \alpha(c) \preceq^\sharp \triangledown^\sharp A$$
$$\amalg^\sharp_{i\in I} c_i \text{ exists} \wedge \triangledown^\sharp_{i\in I} a_i \text{ exists} \wedge \forall i \in I\colon \alpha(c_i) \preceq^\sharp a_i \Rightarrow \alpha\left(\amalg^\sharp_{i\in I} c_i\right) \preceq^\sharp \triangledown^\sharp_{i\in I} a_i.$$

Consequently if the concrete iteration sequence and abstract iteration with widening are convergent then their limits $F^{\natural\epsilon}$ and $F^{\sharp\uparrow\varepsilon}$ are such that $\alpha(F^{\natural\epsilon}) \preceq^\sharp F^{\sharp\uparrow\varepsilon}$.

In fact, that limit might be a post-fixpoint, in which case the result can be refined using a narrowing operator [4]. For more results and details on abstract interpretation, see [17].

3.3 Using TDGs

Basically, TDGs can be used to encode the data handled by the abstract interpretation. Let's call β the encoding between $\mathcal{P}^\sharp$ and $\mathcal{B}$, $F^\flat$ the operator induced by the abstract operator. Considering the properties of TDGs described in section 2 —i.e. their compactness and the efficiency of their operators— the replacement of the abstract iteration by the iteration of $F^\flat$ on $\mathcal{B}$ will in general fill considerably less space, and hopefully take less time than the iteration on classical representations. But, while it is theoretically always possible to find an encoding, not all encodings have these properties. As a trivial example, a coding that associates a variable (and whatever function from $\{0, 1\}$ to $\{0, 1\}$) to each element of $\mathcal{P}^\sharp$ will just fill more space.

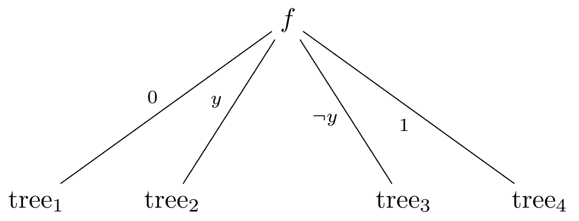
[6] that is to say the limit of the $F^{\sharp\lambda}$.

Although we have no general rule to find a good encoding, we provide some generic tools that can help the design or the use of such an encoding. The first tool will transform encodings of first order functions into encodings of higher order functions. This tool makes the design of the encoding easier, because the encoding of first order functions only is needed. Moreover, it applies to the encoding of the abstract function itself into TDGs. The second tool is a widening operator taking advantage of the structure of the TDGs. It can be used in any abstract interpretation to produce approximations based on the complexity only.

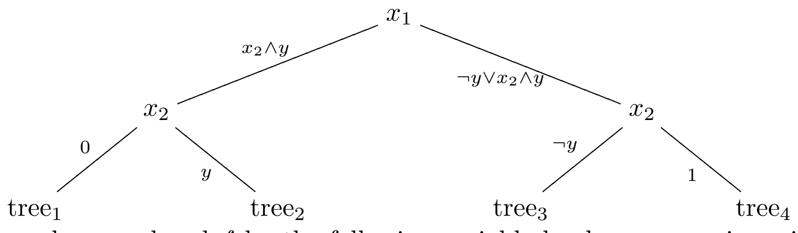
Lifting an Abstract Interpretation.

Informal Presentation. Given the abstractions over two domains, it is possible to abstract the set of functions over those two domains by using the set of functions over the associated abstract domains. If those two domains are already encoded into BDDs, it is then possible to code the functions over those domains using BDDs. This cannot be straightforward, as functions over BDDs are not boolean functions. The point in transforming these functions into BDDs is to replace the variables representing BDDs by (more) boolean variables.

Bounding the number of possible BDDs the variable can represent is a necessary condition to achieve that transformation. So we choose to bound the number of variables of the BDDs that the variable represents. For example, we will work under a limit of one boolean variable for the BDD variable f. For a better understanding, let us come back to Shannon trees —the same can be done with BDDs—. We can represent the function that takes f and gives a boolean expression almost like a Shannon tree. The difference is that, there being four different boolean expression with at most one boolean variable, one should have four subtrees coming from f. The tree will have the following structure:



This variable over BDDs can be replaced by two boolean variables, x_1 and x_2, chosen to be taken before any variable in the subtrees:



So we have replaced f by the following variable boolean expression with at most one boolean variable y: $\neg y \wedge x_1 \vee y \wedge x_2$. This construction will be extended and justified in the next paragraph.

Technical Aspect. Let $\mathcal{P}_1^\sharp$ and $\mathcal{P}_2^\sharp$ be two abstract semantics encoded into TDGs by β_1 and β_2. Moreover we will suppose that $\mathcal{P}_1^\natural(\preceq_1^\natural) \overset{\gamma_1}{\underset{\alpha_1}{\leftrightarrows}} \mathcal{P}_1^\sharp(\preceq_1^\sharp)$ and $\mathcal{P}_2^\natural(\preceq_2^\natural$ $) \overset{\gamma_2}{\underset{\alpha_2}{\leftrightarrows}} \mathcal{P}_2^\sharp(\preceq_2^\sharp)$ are Galois connections[7]. As suggested in [2], such Galois connections can be lifted to functions:

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

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

The lifted semantic domain contains functions from $\mathcal{P}_1^\sharp$ to $\mathcal{P}_2^\sharp$. It means that if we want to extend directly the encoding to the lifted domain, we will need functions over boolean functions, which are not directly representable by TDGs. This is because we cannot make a binary choice after testing a functional variable, as such a variable can take more than two different values. A solution is to transform the tests of functional variables into a sequence of binary tests in required number.

But a variable representing a TDG could take an infinite number of value, as $\mathcal{B}$ is infinite. Accordingly, we will first restrain the set encoded into TDGs to $B^\uparrow \overset{\text{def}}{=} \bigcup_n B_n^\uparrow$, where $B_n^\uparrow \overset{\text{def}}{=} (\{0,1\}^n \to \{0,1\}) \to \mathcal{B}$, $\mathit{Var}_n^\uparrow$ being the set of variables used in $B_n^\uparrow$ and $\mathit{Var}^\uparrow \overset{\text{def}}{=} \bigcup_n \mathit{Var}_n^\uparrow$.

Let $\lambda f.G \in B^\uparrow$; then $\exists n,\ f \in \mathit{Var}_n^\uparrow$, so that testing a value of f can be replaced by testing the value of a finite set of binary variables. Three steps will occur when transforming this expression into a boolean expression: first create this set of binary variables (using $v(f)$), second link an assignment to this set of variable to an assignment to f (using $\mathit{build}(v(f))$), at last replace f in $\lambda f.G$ by the variable function just built. To understand those stages better, we will go through them on a simple example, $\lambda f.\lambda x.fx$[9]. In this example, $f \in \mathit{Var}_1^\uparrow$.

For the construction of the set of boolean variables, we use Shannon's expansion theorem in the following form: a variable f of $\mathit{Var}_{n+1}^\uparrow$ is equivalent to a pair of variables $(\mathit{else}_n(f), \mathit{then}_n(f)) \in \mathit{Var}_n^\uparrow \times \mathit{Var}_n^\uparrow$, where $\mathit{else}_n(f)$ represents the value of f when its first variable is false, and $\mathit{then}_n(f)$ when it is true. As we want to ensure that those variables are distinct, we require the following properties for then_n and else_n: $\forall f, g \in \mathit{Var}_{n+1}^\uparrow$, $\mathit{then}_n(f) \neq \mathit{else}_n(g)$ and both then_n and else_n are injections. We can now define the set of variables associated with a variable f of $\mathit{Var}^\uparrow$, $v(f) \in \wp(\mathit{Var})$:

$$\begin{aligned} v(f) &\overset{\text{def}}{=} \{b(f)\} \quad \text{if } f \in \mathit{Var}_0^\uparrow \\ v(f) &\overset{\text{def}}{=} v(\mathit{else}_n(f)) \cup v(\mathit{then}_n(f)) \quad \text{if } f \in \mathit{Var}_{n+1}^\uparrow \end{aligned}$$

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

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

[9] To distinguish between functional variables and elements of *Var*, elements of *Var* are noted $x, y, x_i, \ldots$, and functional variables $f, g, \ldots$

where b is a bijection from $Var_0^{\uparrow}$ to Var. It is easy to prove by induction that $v(f)$ is just a set of 2^n distinct boolean variables, $\{x_1, \ldots, x_{2^n}\}$. Let us go back to the simple example, $v(f) = \{x_1, x_2\}$, with $x_2 \neq x_1$. Actually, two boolean variables are exactly what is needed to represent the four different possible values of f.

Now we build the variable function associated to this set of boolean variables, so that we can apply this set to boolean values.

$$build\{x\} \overset{\text{def}}{=} x$$

$$build\{x_1, \ldots, x_{2^n}\} \overset{\text{def}}{=} \lambda y.\neg y \wedge build\{x_1, \ldots, x_{2^{n-1}}\} \vee y \wedge build\{x_{2^{n-1}+1}, \ldots, x_{2^n}\}$$

Once again, this definition is justified by Shannon's expansion theorem. In our example, $build\{x_1, x_2\} = \lambda y.\neg y \wedge x_1 \vee y \wedge x_2$.

It is now easy to translate the assignment of a variable f of $Var_n^{\uparrow}$ by $F \in (\{0,1\}^n \to \{0,1\})$ into an assignment of $v(f)$: just assign to each variable of $v(f)$ the value of F applied to the correct boolean values, such that F equals $build(v(f))$ in which all variables of $v(f)$ have been instantiated. So, for example, substituting the variable f of $Var_1^{\uparrow}$ by the function $\lambda x.\neg x$ is the same as substituting $v(f) = (x_1, x_2)$ by $(1, 0)$.

We can now code $B^{\uparrow}$. Let $\lambda f.G \in B^{\uparrow}$, then $\exists n,\ f \in Var_n^{\uparrow}$. Let $\{y_1, \ldots, y_{2^n}\} = s(v(f))$ where s is a permutation on Var such that y_{2^n} is less (for the order on Var) than the smallest possible variable appearing in G. Then if the encoding is called $\beta^{\uparrow}$:

$$\beta^{\uparrow}(\lambda f.G) \overset{\text{def}}{=} \lambda y_1, \ldots, y_{2^n}.G[f/build\{y_1, \ldots, y_{2^n}\}]$$

Example: $\beta^{\uparrow}(\lambda f.\lambda x.fx) = \lambda x_1.\lambda x_2.\lambda x.\neg x \wedge x_1 \vee x \wedge x_2$.

We now have an encoding of $\mathcal{P}^{\sharp}_{1\to 2}$, if we can code it into $B^{\uparrow}$. To achieve this, we will assume the following hypothesis on β_1: for all variable of $\mathcal{P}_1^{\sharp}$ there exists a N such as each 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

$$\beta_{1\to 2}(\lambda f.G) \overset{\text{def}}{=} \beta^{\uparrow}(\lambda \beta_1^v(f).\beta_2(G[f/\beta_1^v(f)]))$$

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 $F_{1\to 2}(\lambda x.y) \overset{\text{def}}{=} \lambda x.F_2(y)$. So if $F_2^{\sharp}$ is coded into TDG, $F^{\sharp}_{1\to 2}$ can be coded into TDG too.

In the particular case where $\mathcal{P}_2^{\sharp} = \mathcal{P}_1^{\sharp}$, we have coded functions over $\mathcal{P}_1^{\sharp}$. As abstract functions are just functions over $\mathcal{P}_1^{\sharp}$, we can thus code them into TDG, making the iteration faster[10]. To encode higher order functions on $\mathcal{P}_1^{\sharp}$, we just have to iterate this construction, as now first order functions are just TDGs. For example, the second order function $\lambda g(\lambda f \lambda x.g(f(x)))$ can be encoded the following way: $g \in Var_1^{\uparrow}$, so $v(g) = \{z_1, z_2\}$ and so $\beta^{\uparrow}(\lambda g(\lambda f \lambda x.g(f(x)))) = \lambda(z_1, z_2, x_1, x_2, x).\neg(\neg x \wedge x_1 \vee x \wedge x_2) \wedge z_1 \vee (\neg x \wedge x_1 \vee x \wedge x_2) \wedge z_2$.

[10] This is not the case if the entire abstract function is not needed. In the case of chaotic iteration for example, we can find better encoding of abstract functions.

A Widening Operator on TDGs. The question of the size of a TDG is at the core of efficiency. Of course taking smaller space is efficient in itself, but as seen in section 2, the speed of operators upon a TDG depends directly on its size. To reduce the size we can use less powerful representations without explicit variables and try out different ordering for the variable. So far however, no really satisfactory solution have been brought out, and some cases will always remain exponential for any ordering. So the proposed solution —specific to abstract interpretation— is a widening operator based on the size of the TDG. This widening operator is very general and can be used whenever 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 long for practical manipulation. It is possible by the use of this widening operator to chose an approximate solution that is compact enough for representation on a computer. This widening is quite different from classical widenings used in abstract interpretation as it does not use any semantic information to approximate the result, but only tries to approximate what fills the most space, leaving as much information as possible in the computation framework.

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 of the TDGs. In fact what the widening operator exactly needs 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 way, the better.

Then, the widening operator takes in a limit size l and a TDG f. The result $\bigtriangledown(l, f)$ is a TDG g such that $|g| \leq l$[11] and $f \preceq^\flat g$. To make sure that it is always possible (for all positive l), we set $(+, 1)$ or $(-, 1)$ as 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: $\bigtriangledown^\flat A \overset{\text{def}}{=} \bigtriangledown(l(\max(A)), \max(A))$ where $\max(A)$ is, if it exists, the maximum of A for the computational ordering[12] ($\sqsubseteq^\flat$), and l a function that yields the limit.

If the abstract function is coded into TDG too, then this widening operator can be used to do static widening by approximating the abstract function. It can be very profitable because if the TDG used to represent the abstract function is too big, each step of the iteration will be too long, and sometimes the size of the TDG representing the iterates will be directly related to the size of the TDG representing the abstract function. Approximating the abstract function is sound, as justified by the following property:

Property 1 *Let F_1 and F_2 be 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*

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

[11] $|g|$ is the number of nodes of g, i.e. its “size”

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

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 follows by induction on the iterates. $\square$

Algorithm. The problem is that for this widening operator we will have to find the best possible g such that $|g| \leq l$, in a decent amount of time. It is not reasonable to search for the best solution[13] as it would theoretically require to explore all the possible derivations of a given TDG, which is exponential in the size of the TDG.

Hence we will try to modify the TDG in order to apply one of the reduction steps described in section 2. To obtain **sharing**, we just consider 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. Because of the properties required on $\preceq^\flat$, this operation gives a TDG greater (for $\preceq^\flat$) than the previous one.

The algorithm proceeds by steps: each step, if the size of the TDG is above the limit, try each of the reductions above and take the best one; repeat. The best one is the one with the highest rate

$$\frac{\text{number of nodes above the limit gained}}{\text{cost of the reduction}}$$

where the cost of the reduction is, for a sharing of T_1 and T_2:

$$\text{cost}(T_1 \to T') \times \text{mult}(T_1) + \text{cost}(T_2 \to T') \times \text{mult}(T_2)$$

and for an elimination of $T = N(x, T_1, T_2)$:

$$(\text{cost}(T_1 \to T') + \text{cost}(T_2 \to T')) \times \text{mult}(T).$$

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 yield $\text{cost}(T_1 \to T')$ and $\text{cost}(T_2 \to T')$ [14]. Mult is the multiplicity of the node, namely the number of time the node would appear in the Shannon tree representation of the TDG, so that changing a node shared by many would cost more than changing one used by only one.

Each forced reduction will not automatically reduce the size of the TDG because the least upper bound may contain more new nodes than gained through the reduction. However, if the size of the TDG is greater than 1, it will contain a node of the form $N(x, (-, 1), (+, 1))$. This is because it is the only possible TDG with one variable, so the only possible node in which the greatest (for $<^v$) variable of the TDG appears. So, if $(+, 1)$ or $(-, 1)$ represents the top of $\mathcal{P}^\sharp$, the reduction of this node into the top will always be tried, ensuring that at least

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

[14] 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'.

one of the modifications tried on the TDG within one step reduces the size of the graph. Thus if the limit is positive the size of the TDG will at each step either decrease or be less than or equal to the limit. Besides, after each step, the new TDG is greater than the previous one for $\preceq^{\flat}$, so the algorithm is correct.

Example Consider the function $f = (y \wedge x) \vee (x \wedge \neg z) \vee (z \wedge \neg x)$ defined in the examples of BDDs, with the pointwise ordering for $\preceq^{\flat}$ based on $0 \preceq^{\flat} 1$. See figure 3 for the possible solutions.

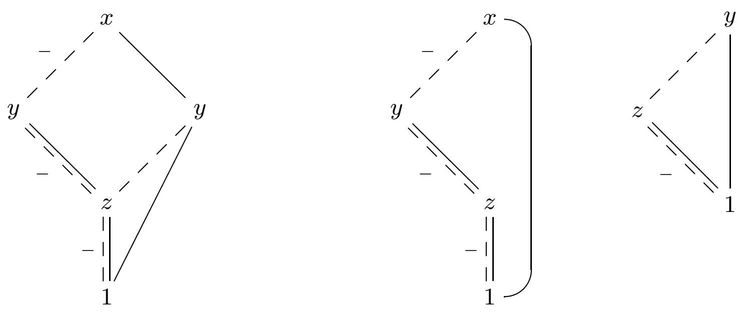


Fig. 3. f and the two best approximates with 3 nodes or less.

Complexity. Considering that at each step, the size of the TDG is reduced by one at least, the number of steps is smaller than the difference between the limit and the size of the TDG. But this is still too much: this difference may be exponential. To reach faster a size closer to the limit, we use a less refined algorithm which assumes that $(+, 1)$ or $(-, 1)$ represents the top. For each node such that the size of the TDG without that node [15] lies between the limit and the limit plus half the difference between the limit and the size of the TDG, we try to replace it by the top and take the one that gives the best result. That way, each step of this algorithm at least halves the difference between the limit and the size of the TDG.

The most precise algorithm requires each pair of node to be tested. The multiplicity of each node can be calculated in a time polynomial in the number of nodes by going through the TDG and tagging the nodes. As a result, if the computation of the least upper bound (plus the computation of the costs) is polynomial in the size of the TDGs, then the most precise algorithm is polynomial in the size of the TDG.

Thus we can combine the two algorithms in a way 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 use the other one. 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.

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

4 A Complete Example: Strictness Analysis

In this section we describe a complete example of program analysis using abstract interpretation and TDGs. Let's define first the property to be computed.

Definition 2. *A function f is said to be* strict *in one of its arguments x if everytime the evaluation of that argument fails, the evaluation of f(x) fails*

The evaluation fails if it ends with an error or 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 compilar optimization of a call-by-need prgramming language. The principle of such an implementation is to keep the arguments of functions in a closure until they are first needed in the evaluation of the function and then evaluate them. If a function is strict in an argument then that argument will be always needed, so the compiler can evaluate the argument anyway[16], saving in the meantime the space for the closure.

Strictness analysis is a good example of application of TDGs because it is a useful analysis —in a compiler for example— but the most 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 which will be coded into TDGs. The well-known analysis we use as a basis is one developed by Alan Mycroft in [5], that still seems to be one of the most precise, and that has the advantage of being already coded into boolean functions.

The Concrete Domain. Mycroft's analysis deals with 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 $\mathcal{D}$ [19] where $\mathcal{D}$ is a complete domain with infimum $\perp$ and the values from the base types, such as integers. $\perp^\natural \overset{\text{def}}{=} \lambda x.\perp$. The concrete semantic function is constructed by induction on the syntax of the expression defining the function: $F^\natural = S[\![\texttt{f(x)=}e]\!]$.

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

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

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

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

[16] Assuming there is no side effect.

The Abstract Domain. The abstract domain introduced by Mycroft is the set of monotonic functions from $\{0,1\}$ to $\{0,1\}$, with the ordering $0 \preceq^\sharp 1$[17], 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.

$\bot^\sharp \overset{\text{def}}{=} \lambda x.0$. The abstract semantic function is also defined by induction on the syntax:

$$\begin{aligned}
S^\sharp[\![\texttt{f(x)=b(}e_1\texttt{,...,}e_n\texttt{)}]\!](g^\sharp) &\overset{\text{def}}{=} b^\sharp(S^\sharp[\![\texttt{f(x)=}e_1]\!](g^\sharp),\ldots,S^\sharp[\![\texttt{f(x)=}e_n]\!](g^\sharp))\\
S^\sharp[\![\texttt{f(x)=x}]\!](g^\sharp) &\overset{\text{def}}{=} \lambda x.x\\
S^\sharp[\![\texttt{f(x)=f(}e\texttt{)}]\!](g^\sharp) &\overset{\text{def}}{=} g^\sharp \circ S^\sharp[\![\texttt{f(x)=}e]\!](g^\sharp)
\end{aligned}$$

$b^\sharp$ 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, $\mathcal{P}^\natural \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \mathcal{P}^\sharp$:

$$\begin{aligned}
\alpha(f)(0) &\overset{\text{def}}{=} \text{if } \{x|(\bot,x)\in f\} = \{\bot\} \text{ then } 0 \text{ else } 1\\
\alpha(f)(1) &\overset{\text{def}}{=} \text{if } \{y|x\in\mathcal{D}\wedge(x,y)\in f\} = \{\bot\} \text{ then } 0 \text{ else } 1
\end{aligned}$$

$$\begin{aligned}
\gamma(\lambda x.0) &\overset{\text{def}}{=} \lambda x.\bot\\
\gamma(\lambda x.x) &\overset{\text{def}}{=} \{(\bot,\bot)\} \cup ((\mathcal{D}-\{\bot\})\times\mathcal{D})\\
\gamma(\lambda x.1) &\overset{\text{def}}{=} \mathcal{D}\times\mathcal{D}
\end{aligned}$$

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

Proof. By induction on the syntax, we shall first prove that $\forall f \in \mathcal{P}^\natural$ and $\forall g^\sharp \in \mathcal{P}^\sharp, \alpha(f) \preceq^\sharp g^\sharp$ implies that $\alpha(F(f)) \preceq^\sharp F^\sharp(g^\sharp)$, then as $\alpha(\bot^\natural) = \bot^\sharp$ the inequation on the fixpoints will follow by induction on the iterates.

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

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

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

The first line is given by definition of S, the second by hypothesis of induction the third by the property of the abstract constants, and finally the fourth by definition of $S^\sharp$.

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

For the last step of the proof we need a few more results on the composition of relations. $R_1 \circ R_2 \overset{\text{def}}{=} \{(x,w) | (x,v) \in R_2 \wedge (v,w) \in R_1\}$. Suppose $\alpha(R_1) \circ \alpha(R_2)(a) = 0$. If $\alpha(R_2)(a) = 0$ then $\{y | x \in A \wedge (x,y) \in R_2\} = \{\bot\}$[18] 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 $\forall R_i$, $\alpha(R_1 \circ R_2) \preceq^\sharp \alpha(R_1) \circ \alpha(R_2)$.

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

The first line is the definition of S. Then use 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, $\alpha(S[\![\texttt{f(x)=}e]\!](f)) \preceq^\sharp S^\sharp[\![\texttt{f(x)=}e]\!](g^\sharp)$ 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 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 also strict.

4.2 The Encoding

To code the abstract domain, we merely add variable names and $\mathcal{P}^\flat$ becomes $\mathcal{B}_1$. Abstract functions could be coded using the method presented in the previous

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

section, as they are functions from $\mathcal{B}_1$ to $\mathcal{B}_1$. The problem when dealing with higher order functions is that, since the size of the type is increasing and each step of the iteration requires every possible value of the previous iterate, we will lose all the interest of the TDG for recursive functions. Accordingly we prefer to code each recursive call by a new variable, keeping the arguments of the recursive call. That way, each step of the iteration will 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 easily be 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: $\mathit{build}\{x_1, \ldots, x_{2^n}\} \overset{\text{def}}{=} \lambda y. \wedge \mathit{build}\{x_1, \ldots, x_{2^{n-1}}\} \vee y \wedge \mathit{build}\{x_{2^{n-1}+1}, \ldots, x_{2^n}\}$.

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

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

ρ is an environment function. It maps program variables 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, which is, if f is such a function, $\mathit{build}(v(f))$. We use the type of the variable in order to know what $v(f)$ is, that is to say the exact number of boolean variables needed. 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)`[19].

The type of `x` is $\alpha \rightarrow \beta \rightarrow \gamma$, and so it can take at least 2^4 different values. So we need four boolean variables $v(\mathtt{x}) = \{x_1, x_2, x_3, x_4\}$ to represent all the different possible states of `x`.

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

$\mathrm{lfp}(S^\flat[\![\mathtt{(x\ z)(y\ z)}]\!]\rho)$ 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 us 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

[19] This function is one of the most famous higher ordre functions as with `s`, `k` (`k x y = x`) and `i` (`i x = x`), one can code the entire λ-calculus.

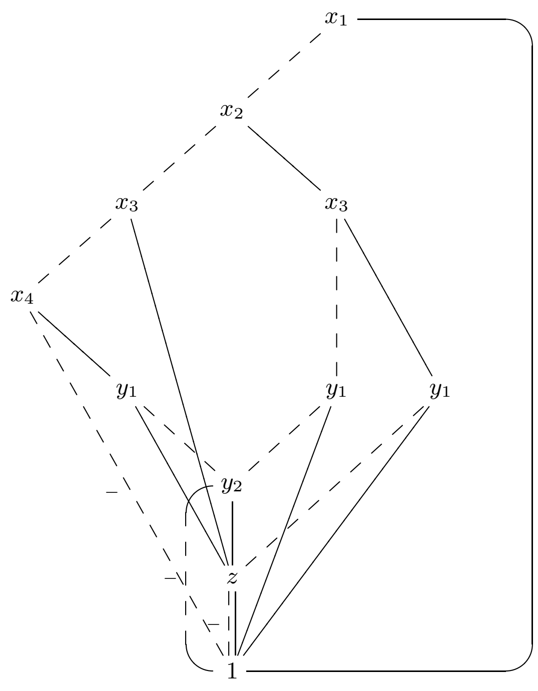


Fig. 4. `s x y z = (x z)(y z)`

function, so that the same widening can be applied to the abstract set and to the abstract functions.

4.3 Practical Results

Strictness analysis have been implemented using TDGs[20], 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 below are for the interpreted version (in camllight) and could be improved by compilation. Besides the implementation of Sebastian Hunt was only a prototype implementation, so the comparison might be unfair. However, this results should not be taken as comparable with state of the art strictness analysis, but as an indication of what can be gained using TDGs in program analysis.

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 problems raised by these examples are typical and standard for strictness analysis. The first three examples didn't require the use of the widening operator, so the results have the same accuracy as with frontiers. The first one 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[21]. The third one uses foldr at a higher order, applying it to append, so the result of the analysis is much bigger.

The fourth example analyzes foldr written in continuation passing style, leading to a drastic increase in the type order. Two functions are analyzed with type $(\alpha\ \texttt{list} \rightarrow \alpha\ \texttt{list} \rightarrow (\alpha\ \texttt{list} \rightarrow \beta) \rightarrow \beta) \rightarrow \alpha\ \texttt{list list} \rightarrow \alpha\ \texttt{list} \rightarrow (\alpha\ \texttt{list} \rightarrow \beta) \rightarrow \beta$. 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 [23], that gives results in a few seconds. However, this analysis only answers one question and so is not usable for separate compilation. Moreover, the same technique could be applied using TDGs to answer the same question.

5 Conclusion

This approach proved to be efficient in strictness analysis and could be advantageously used in many other abstract interpretations, whatever the context, as its

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

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

idea is based on the semantic domain not on a fixpoint algorithm. For instance it would work with backward and forward analysis, total or partial fixpoint computation, etc. 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 in association with other works on this subject. The idea of lazy evaluation of abstract functions from Ferguson and Hughes was mentioned above, but the results of Baraki[22] on interpretation of polymorphic functions in [22] 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.

Acknowledgments

I would particularly like to thank Patrick Cousot, for his very heplful advice and support, and for the good idea of using TDGs to represent abstract properties. All the members of his team helped me too with every little problem. The comments from Thomas Jensen were very useful to increase the clarity of this paper. Thanks also to Sebastian Hunt 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 Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*. 1979.
5. A. Mycroft. The Theory and Practice of Transforming Call-by-Need into Call-by-Value.*Proceedings of the Fourth International Symposium on Programming, LNCS 83* pp 270–280. 1980.
6. R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput. C-35*, pp. 677–691. 1986.
7. G. L. Burn, C. Hankin & S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming 7*, pp. 249–278. 1986.
8. P. Hudak & J. Young. Higher Order Strictness Analysis in Untyped Lambda Calculus. *ACM Principles of Programming Languages* pp 97–109. 1986.
9. J. P. Billon. Perfect Normal Forms for Discrete Programs. *Technical report 87039 BULL*. 1987.
10. J. C. Madre & J. P. Billon. Proving Circuit Correctness Using Formal Comparison between Expected and Extracted Behavior. *Proc. of the 25th DAC*. 1988
11. A. R. Brayton, B. Lin & H. J. Touati. Don't Care Minimization of Multi-Level Sequential Logic Network. *Proc. of ICCAD'90*. 1990.

[22] He designed a way of using results on polymorphic functions to find the properties of their instantiations

12. J. C. Madre, C. Berthet & O. Coudert. New Ideas in Symbolic Manipulation of Finite State Machines. *Proc. of ICCAD'90*. 1990.
13. J. Schwable & K. L. McMillan. Formal Verification of the Encore Gigamax Cache. *International Symposium on Shared Memory Multiprocessor*. 1991.
14. D. Taubner, E. Enders & T. Filkorn. Generating BDDs for Symbolic Model Checking in CCS. *Proc. of CAV'91*, pp. 203–213. 1991.
15. H. J. Touati, H. Savoj & R. K. Brayton. Extracting Local Don't Care for Network Optimization. *Proc. of ICCAD'91*. 1991.
16. R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys, Vol. 24*. pp. 293–318. 1992.
17. P. Cousot & R. Cousot. Abstract Interpretation Framework. *Journal of Logic and Computation, vol 2, No 4*, pp. 511–547. 1992.
18. C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE. *Thse de l'Université de Grenoble 1*, chap 11. 1992.
19. P. Cousot & R. Cousot. Galois Connection Based Abstract Interpretations for Strictness Analysis. *Proceedings of the International Conference on Formal Methods in Programming and their Applications, LNCS 735* pp 98–127. 1993.
20. B. Le Charlier & P. Van Hentenryck. Groundness Analysis for Prolog: Implementation and Evaluation of the Domain *Prop. Proc. of PEPM'93*. 1993
21. M.-M. Corsini, M. Musumbu, A. Rauzy & B. Le Charlier Efficient Bottom-up Abstract Interpretation of Logic Programs by means of Constraint Solving. *PLILP '93*. 1993.
22. G. Baraki. Abstract Interpretation of Polymorphic Higher-Order Functions. (PhD thesis) *Computing Science research report of the University of Glasgow*. 1993.
23. A. Ferguson & J. Hughes. Fast Abstract Interpretation Using Sequential Algorithms. *Proc. of WSA '93*, pp. 45–59. 1993.
24. P. Van Hentenryck, A. Cortesi & B. Le Charlier. Evaluation of *Prop. Journal of Logic Programming*. pp 237–278. 1995.