A Scalable Segmented Decision Tree Abstract Domain

Patrick Cousot[2,3], Radhia Cousot[1,3], and Laurent Mauborgne[3,4]

[1] Centre National de la Recherche Scientifique
[2] Courant Institute of Mathematical Sciences, New York University
[3] École Normale Supérieure, Paris
[4] Instituto Madrileño de Estudios Avanzados, Madrid

Dedicated to the memory of Amir Pnueli

1 Introduction

The key to precision and scalability in all formal methods for static program analysis and verification is the handling of disjunctions arising in relational analyses, the flow-sensitive traversal of conditionals and loops, the context-sensitive inter-procedural calls, the interleaving of concurrent threads, etc. Explicit case enumeration immediately yields to combinatorial explosion. The art of scalable static analysis is therefore to abstract disjunctions to minimize cost while preserving weak forms of disjunctions for expressivity.

Building upon packed binary decision trees to handle disjunction in tests, loops and procedure/function calls and array segmentation to handle disjunctions in array content analysis, we introduce *segmented decision trees* to allow for more expressivity while mastering costs via widenings.

2 Semantic Disjunctions in Abstract Interpretation

The main problem in applying abstract interpretation [2,5,6] to static analysis is to abstract a non-computable fixpoint collecting semantics $\mathbf{lfp}^{\sqsubseteq}_{\perp} F$ for a concrete transformer $F \in \mathcal{C} \mapsto \mathcal{C}$, partial order $\sqsubseteq$, and infimum $\perp$ into an abstract semantics $\mathbf{lfp}^{\sqsubseteq^\sharp}_{\perp^\sharp} F^\sharp$ for an abstract transformer $F^\sharp \in \mathcal{A} \mapsto \mathcal{A}$, abstract order $\sqsubseteq^\sharp$, and abstract infimum $\perp^\sharp$ where the existence of fixpoints is guaranteed by Tarski's theorem [21] on complete lattices or its extension to complete partial orders (cpos). The collecting semantics is the specification of the undecidable properties we want to collect about programs. The abstract semantics is an effective approximation of the collecting semantics. For soundness, $\mathbf{lfp}^{\sqsubseteq}_{\perp} F \sqsubseteq \gamma(\mathbf{lfp}^{\sqsubseteq^\sharp}_{\perp^\sharp} F^\sharp)$ where $\gamma \in \mathcal{A} \mapsto \mathcal{C}$ is the concretization function. Particular cases involve a Galois connection $\langle \mathcal{C}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq^\sharp \rangle$ such that $\forall x \in \mathcal{C} : \forall y \in \mathcal{A} : \alpha(x) \sqsubseteq^\sharp y \iff x \sqsubseteq \gamma(y)$ and the case of completeness requiring $\alpha(\mathbf{lfp}^{\sqsubseteq}_{\perp} F) = \mathbf{lfp}^{\sqsubseteq^\sharp}_{\perp^\sharp} F^\sharp$.

In general the concrete domain $\langle \mathcal{C}, \sqsubseteq, \perp, \sqcup \rangle$ is a complete lattice or cpo and the concrete transformer F is in disjunctive form $F \triangleq \bigsqcup_{i \in \Delta} F_i$ and often

Z. Manna and D. Peled (Eds.): Pnueli Festschrift, LNCS 6200, pp. 72–95, 2010.
© Springer-Verlag Berlin Heidelberg 2010

completely distributive ($\forall i \in \Delta : F_i(\bigsqcup_{j \in \Delta'} X_j) = \bigsqcup_{j \in \Delta'} F_i(X_j)$) or continuous (completely distributive on increasing chains).

In that most usual case, the iterative fixpoint computation $X^0 = \bot, \ldots, X^{n+1} = F(X^n), \ldots, X^\omega = \bigsqcup_{n \geqslant 0} X^n = \mathbf{lfp}_\bot^\sqsubseteq F$ is $X^1 = F(\bot) = \bigsqcup_{i \in \Delta} F_i(\bot)$, $X^2 = F(X^1) = \bigsqcup_{i \in \Delta} F_i(\bigsqcup_{j \in \Delta} F_j(\bot)) = \bigsqcup_{i,j \in \Delta^2} F_i \circ F_j(\bot), \ldots, X^{n+1} = F(X^n) = \bigsqcup_{i \in \Delta} F_i(\bigsqcup_{i_1,\ldots,i_n \in \Delta^n} F_{i_1} \circ \ldots \circ F_{i_n}(\bot)) = \bigsqcup_{i_1,\ldots,i_n,i_{n+1} \in \Delta^{n+1}} F_{i_1} \circ \ldots \circ F_{i_n} \circ F_{i_{n+1}}(\bot), \ldots$, so that passing to the limit $\mathbf{lfp}_\bot^\sqsubseteq F = X^\omega = \bigsqcup_{n \geqslant 0} X^n = \bigsqcup_{n \geqslant 0} \bigsqcup_{i_1,\ldots,i_n \in \Delta^n} F_{i_1} \circ \ldots \circ F_{i_n}(\bot)$. This shows that the disjunctive explosion problem appears in the concrete iterative fixpoint definition.

If the abstraction is a Galois connection, the abstraction preserves existing joins. It follows that $\alpha(\mathbf{lfp}_\bot^\sqsubseteq F) = \alpha(\bigsqcup_{n \geqslant 0} \bigsqcup_{i_1,\ldots,i_n \in \Delta^n} F_{i_1} \circ \ldots \circ F_{i_n}(\bot)) = \bigsqcup^\sharp_{n \geqslant 0} \bigsqcup^\sharp_{i_1,\ldots,i_n \in \Delta^n} \alpha(F_{i_1} \circ \ldots \circ F_{i_n}(\bot))$ which is most often over approximated as $\bigsqcup^\sharp_{n \geqslant 0} \bigsqcup^\sharp_{i_1,\ldots,i_n \in \Delta^n} \alpha \circ F_{i_1} \circ \gamma \circ \alpha \circ F_{i_2} \circ \gamma \circ \ldots \circ \alpha \circ F_{i_n} \circ \gamma(\alpha(\bot))) \sqsubseteq^\sharp \bigsqcup^\sharp_{n \geqslant 0} \bigsqcup^\sharp_{i_1,\ldots,i_n \in \Delta^n} F^\sharp_{i_1} \circ F^\sharp_{i_2} \circ \ldots \circ F^\sharp_{i_n}(\bot^\sharp)) = \mathbf{lfp}_{\bot^\sharp}^{\sqsubseteq^\sharp} F^\sharp$ where $\forall i \in \Delta : \alpha \circ F_i \circ \gamma \sqsubseteq^\sharp F^\sharp_i$, $F^\sharp \triangleq \bigsqcup^\sharp_{i \in \Delta} F^\sharp_i$ and $\bot^\sharp \triangleq \alpha(\bot)$. This shows that the disjunctive explosion problem does also exist in the abstract.

The situation is even worst in absence of best abstraction, that is of a Galois connection, since the concrete transformers F_i may have many, possibly non-comparable, abstractions $F^\sharp_i$. In absence of minimal abstractions (as shown by the abstraction of a disk by polyhedra [12]), infinitely many potential abstractions may exist. Choosing which abstraction should better be used during the analysis is another source of potential combinatorial explosion.

3 Handling Disjunctions in Abstract Interpretation

Contrary to purely enumerative or symbolic encodings of program properties, abstract interpretation offers solutions to the combinatorial explosion of disjunctions so as to minimize computational costs. The key idea is to abstract away irrelevant properties of the collecting semantics.

The abstract domain $\langle \mathcal{A}, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp \rangle$ can be chosen as finite (e.g. predicate abstraction [3,13]) or better of finite height (e.g. constant propagation [15]) to bound n in $\mathbf{lfp}_{\bot^\sharp}^{\sqsubseteq^\sharp} F^\sharp = \bigsqcup^\sharp_{n \geqslant 0} {F^\sharp}^n(\bot^\sharp))$.

However this solution has been shown to have intrinsic limitations [8] that can be eliminated thanks to infinite abstract domains not satisfying the ascending chain condition together with widenings ∇ and narrowings Δ [2,4,5] (including the common practice of including the widening in the transformers $F^\sharp_i$, $i \in \Delta$ mentioned in [9], by choosing $\boldsymbol{\lambda} X \bullet X \nabla (\alpha \circ F_i \circ \gamma(X)) \sqsubseteq^\sharp F^\sharp_i$, $i \in \Delta$, which is not such a good idea since it precludes the later use of a narrowing).

Moreover, in absence of a best abstraction, that is of a Galois connection, a choice is usually made among the possible non-comparable abstractions $F^\sharp_i$ of the concrete transformers F_i to minimize costs [7].

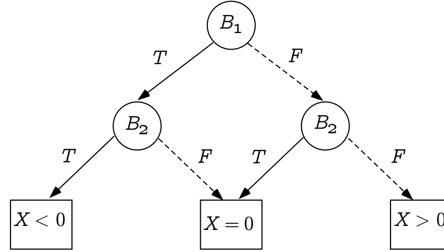
The objective of minimizing computational costs is antagonist to that of precision necessary for making proofs, so that a lot of work in abstract interpretation is on the design of abstract domains offering cost effective yet precise enough ways of abstracting infinite disjunctions.

In this line of work, we consider the examples of binary decision trees in Sect. 4 and segmented arrays in Sect. 6 which common generalization is segmented decision trees as introduced in Sect. 7, a generalization which is more precise while remaining scalable.

4 Binary Decision Trees

Inspired by the use of Binary Decision Diagrams in abstract interpretation [17,18], binary decision trees can express disjunctive properties depending on the values of binary variables with opportunistic sharing at the leaves [16,19]. They are an instance of the reduced cardinal power of abstract domains [6, Sect. 10.2] mapping the values of boolean variables (represented in decision nodes) to an abstraction of the other variables (represented in the leaf nodes) for the values of the boolean variables along the path leading to the leave.

Example 1. The following binary decision tree



written $[\![B_1 : [\![B_2 : (\!| X < 0 |\!), (\!| X = 0 |\!)]\!], [\![B_2 : (\!| X = 0 |\!), (\!| X > 0 |\!)]\!]]\!]$ encodes
$(B_1 \wedge B_2 \wedge X < 0) \vee (((B_1 \wedge \neg B_2) \vee (\neg B_1 \wedge B_2)) \wedge X = 0) \vee (\neg B_1 \wedge \neg B_2 \wedge X > 0)$.
□

The parenthesized representation of trees uses $(\!| \ldots |\!)$ for leaves and $[\![x : \ldots]\!]$ for left to right decision nodes on variable x.

Example 2. In the following example, Astrée [10] discovers a relation between the boolean variable B and the unsigned integer variable X.

```
% cat -n decisiontree.c
     1  typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
     2  BOOLEAN B;
     3  void main () {
     4    unsigned int X, Y;
     5    while (TRUE) {
     6      __ASTREE_analysis_log(());
```

```
     7       B = (X == 0);
     8       if (!B) {
     9         Y = 1 / X;
    10       };
    11     };
    12   }
astree --exec-fn main --print-packs decisiontree.c
...
  <boolean relations:decisiontree.c@9@5=
  if B then <integers (intv+cong+bitfield+set): X in {0} >
  else <integers (intv+cong+bitfield+set): X in [1, 4294967295] >
  >
...
%
```

At line 9, the relation between B and X is expressed by a binary decision tree with B as only decision node and X in the leave nodes. This binary decision tree states that if B is TRUE then X is zero else X is positive. Since B is checked to be false there is no division by zero at line 9. □

5 Variable Packing

Relational abstractions such as octagons [20], polyhedra [12] or binary decision trees in Sect. 4 can express relations between values of variables hence complex disjunctions. However their cost may grow polynomially or even exponentially in size and time. Even for abstract domains such as the octagon domain which has a very light cubic cost compared to many relational domains (such as exponential costs in the number of variables for polyhedra [12]), this would still be too high for very large embedded programs with tens of thousands variables as found in aeronautics.

Variable packing is a way to control memory and computation costs by limiting the number of variables that can be related at a given program point. The decision of which variables can be considered in such abstract relations at each program control point can be taken during the analysis e.g. by widenings (deciding which variables to eliminate by projection from the relation). Variable packing is an even more radical solution which consists in controlling costs a priori, by statically restricting the variables which can be considered in such abstract relations at each program control point. Such a more radical solution is necessary when the cost of even one iteration with all variables in the relation is prohibitive.

The idea is to make small packs of variables that are related while no attempt is made to relate variables appearing in different packs. A simple and cheap pre-analysis groups variables that are interdependent (used together in expressions, loop indices, etc.) in a way that can be represented by the relation. Relations are established among variables associated to each pack, but no relation is kept between variables of distinct packs. The cost thus becomes linear, as it is linear in the number of packs (which is linear in the code size, and so, in the number of variables) and, e.g. for octagons, cubic in the size of packs (which depends on the size of the considered packs, and is a constant).

Example 3. In *Ex. 2*, Astrée has packed the boolean variable B and the unsigned integer variable X together (but the pack does not contain variable Y). The abstract domain used in the leaves is the reduced product [6] of several abstract domains which can be specified as an option of the analysis and by default are the following

```
/* Domains: ... Packed (Boolean relations (based on Absolute value
equality relations, and Symbolic constant propagation (max_depth=
20), and Linearization, and Integer intervals, and Integer
congruences, and Integer finite sets, and Integer bitfields, and
Float intervals)), and ... */
...
Boolean relations :
List of packs
  decisiontree.c@9@5 { B X }
...
  <boolean relations:decisiontree.c@9@5=
  if B then <integers (intv+cong+bitfield+set): X in {0} >
  else <integers (intv+cong+bitfield+set): X in [1, 4294967295] >
  >
...
```

The output in *Ex. 2* only prints informative results, specifically intervals [4,5], simple congruences [14], bit fields (recording an invariant bit pattern in the binary representation of the variable, if any) and sets of small values (recording the set of possible values of the variable in a small range of values near zero) for that example. ☐

Candidates for packing in a binary decision tree are the boolean variables to which a boolean expression is assigned or which are involved in a test as well as the variables which depend directly or indirectly on such a boolean variable, with a maximum number of boolean variables which can be set by an option `--max-bool-var` (3 by default). The option `--print-packs` allows printing packs of variables determined by the pre-analysis for the binary decision trees. In case of insufficient precision, Astrée can be asked to create binary decision tree packs containing given variables by the `__ASTREE_boolean_pack` directive inserted in the program code. Of course putting all boolean program variables in a single pack would certainly achieve great precision but is also the best way to get a combinatorial explosion.

6 Array Segmentation

Array segmentation was introduced in [11] to abstract array content (as opposed to array bounds [4]). The array is divided into consecutive segments. The content of each segment is abstracted uniformly but different segments can have different abstractions. Starting from a single initial segment with uninitialized content, segments are split and filled by assignments to arrays elements and joined when merging control flows. A widening may be necessary to merge segments so as to avoid the degenerescence to the case of one element per segment.

The bounds of the segments are specified by a set of side-effect free expressions which all have the same concrete value (maybe unknown in the abstract). Segments can be empty thus allowing an array segmentation to encode a disjunction of cases in a way that avoids the explosion of cases.

Example 4. The segmentation $\{0\}\top\{1\}0\{i\}?\top\{n\}$ of an array A states that $0 < 1 \leqslant i < n$, that the values of the array elements A[0], A[i], A[i+1], ..., A[n-1] are all unknown, while the values of A[1], A[2], ..., A[i-1], if any, are all initialized to zero. It is possible that $i = 1$ in which case the segment A[1], ..., A[i-1] is empty, and $i < n$ so that the segment A[i], ..., A[n-1] cannot be empty (it contains at least one element).

The segmentation $\{0\}\top\{1\}0\{i\}\top\{n\}$ is similar but for the fact that $i > 1$ so that the segment A[1], ..., A[i-1] cannot be empty. So $\{0\}\top\{1\}0\{i\}?\top\{n\}$ is a compact encoding of the disjunctive information $(\{0\}\top\{1,i\}\top\{n\}) \vee (\{0\}\top\{1\}0\{i\}\top\{n\})$ distinguishing the first case $i = 1$ from the second $i > 1$.

Similarly $\{0\}\top\{1\}0\{i\}?\top\{n\}?$ is the disjunction $(\{0\}\top\{1,i,n\}) \vee (\{0\}\top\{1,i\}\top\{n\}) \vee (\{0\}\top\{1\}0\{i,n\}) \vee (\{0\}\top\{1\}0\{i\}\top\{n\})$. Of course, expressivity is limited since it is not possible to express that either $i = 1$ or $i = n$ but not both (although this might be separately expressible by the abstraction of the simple variables i and n). □

Note that there are no holes in the segmentation since any such hole is just a segment which content is unknown.

An enriched semantics of arrays is used viewing an array value as the pair of an index and the value of the corresponding array element. In this way the uniform abstraction used in a segment can relate array values within each segment to their index included between the segment bounds.

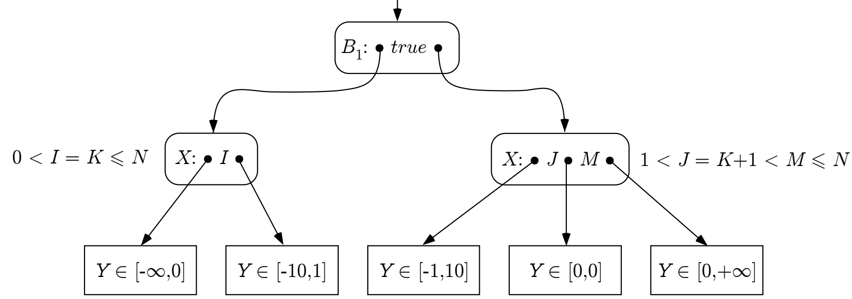
The array segmentation is implemented as a functor which means that the abstract domains representing sets of expressions and array index-elements abstractions should be passed as parameters to the functor to create a particular instance. The advantage of this approach is that the abstract domain parameters can be changed without having to rewrite the static analyzer.

Moreover the static analyzer can optionally perform a reduced product [6] between an instance of the array segmentation functor and the abstract domains that are used for variables appearing in the expressions of the segment bounds. It follows that the array segmentation takes into account both the operations on arrays (array element access and assignment) and operations on variables related to array indexes.

7 Segmented Decision Trees

Segmented decision trees are decision trees where the choices are made on the values of variables according to ranges specified by a symbolic segmentation.

Example 5. The segmented decision tree (where *false* < *true* for booleans)



can be written in the parenthesized form

$$\begin{array}{l}[\![B_1 : \ [\![X\,\{0 < I = K \leqslant N\} : \quad (\!|Y \in [-\infty, 0]\,|\!) \ \ I \ \ (\!|Y \in [-10, 1]\,|\!) \ \ ]\!] \ \ \mathit{true} \\ \qquad [\![X\,\{1 < J = K + 1 < M \leqslant N\} : \\ \qquad\qquad (\!|Y \in [-1, 10]\,|\!) \ \ J \ \ (\!|Y \in [0, 0]\,|\!) \ \ M \ \ (\!|Y \in [0, +\infty]\,|\!) \ \ ]\!] \ ]\!]\end{array}$$

This segmented decision tree encodes the fact that if B_1 is false (i.e. $B_1 < \mathit{true}$) then if $X < I$ then Y is non-positive while if $X \geqslant I$ then $-10 \leqslant Y \leqslant 1$. Similarly, if B_1 is true (i.e. $B_1 \geqslant \mathit{true}$) then either $X < J$ and $-1 \leqslant Y \leqslant 10$, or $J \leqslant X < M$ and Y is null, or $X > M$ and Y is non-negative. So the leaf nodes specify abstract properties of Y while the decision nodes on B_1 and X specifying conditions for these properties to hold. Attached to each decision node, is a side relation on expressions that holds in the concrete under the condition that this node is reached. For example $(B_1 \wedge 1 < J = K + 1 < M \leqslant N) \vee (\neg B_1 \wedge 0 < I = K \leqslant N)$. These expressions are usually in a restricted normal form. In this example the normal form of expressions is an integer constant, a variable, or a variable plus an integer constant and the side relations are those expressible with the octagon abstract domain [20]. The segment bounds are any representative of the equivalent class of expressions which have equal concrete values (so that we could have chosen K for I and $K + 1$ for J). The abstract domain of side relations is assumed to be expressive enough to maintain such equality information between expressions in normal form (i.e. $I = K$ and $J = K + 1$). □

As for boolean decision trees, an ordering is imposed on all decision variables. That allows binary operations on decisions trees to operate on the same variable [1]. But unlike binary decision trees, the number of choices for a given variable is not bounded *a priori* and the choices may be on different criteria (the bounds in the symbolic segmentations) at each node, even if they have the same decision variables.

As for simple array segmentation, the ordering of the bounds of each segment describes an order on expressions. That means that segments could describe two kinds of informations: a serie of tests deciding what choice to make and a pre-order on some expressions. Unlike for array segmentation, that information

[1] In addition it may allow to eliminate the nodes with only one child, an optimization we will not apply in this paper.

would now be relative to the choices above a node in a tree. So we decided to separate the two notions, such that at each node, we have a decision variable, a pre-order and a segmentation that respects the pre-order (instead of prescribing it) and leads to subtrees. A direct consequence is that segments will be much more simple, as that removes the necessity of a `?` tag for emptyness or of the lowest and highest bounds of the segmentation. Also, it allows the use of single expressions as bounds instead of sets of expressions.

Separating the pre-order information from the decision process allows us to use much more precise domains for the pre-order and leads to more precise unifications of segmentations. But storing a pre-order on all possible expressions that can be used at a given node in the tree might lead to very expensive representations. So we chose instead to rely on reduction with other abstract domains for pre-order information valid at every node in the tree and store at each node the ordering information that we can add to what holds for its father. In that way, if the abstraction of an instruction implies an ordering that is not relevant to the tree, it will not increase the complexity of the tree, but further operations might still use that information from reduction with the other abstract domains. In the drawings and examples, we will represent that information as a set of inequations between expressions, but in practice, depending on the canonical expressions we use, we can implement it in a more efficient way, using for example small octagons if the canonical expressions are of the form $\pm X + c$ with X a variable and c a constant. Also, in order to simplify the presentation, in the schemata, we put all global pre-order information at the root[2].

Definition 1. A segmented decision tree $t \in \mathbb{T}((\mathbb{D}, <_{\mathbb{D}}), \mathbb{E}, D_c, D_\ell)$ over decision variables in the totally ordered set $(\mathbb{D}, <_{\mathbb{D}})$, canonical expressions in $\mathbb{E}$, ordering abstract domain D_c (with concretization γ_c) and leaf abstract domain D_ℓ (with concretization γ_ℓ) is either $(\!| p |\!)$ with p an element of D_ℓ or $[\![\mathtt{x}\,\{C\} : t_0 b_1 t_1 \dots b_n t_n]\!]$ such that $\mathtt{x}$ is the smallest variable in $\mathbb{D}$, each b_i ($1 \leqslant i \leqslant n$) is an element of $\mathbb{E}$, C is an element of D_c and each $t_i \in \mathbb{T}((\mathbb{D}\backslash\{x\}, <_{\mathbb{D}}), \mathbb{E}, D_c, D_\ell)$ ($0 \leqslant i \leqslant n$). □

To define the concretization of a segmented decision tree, we will write ρ for concrete environments assigning concrete values $\rho(\mathtt{x})$ to variables $\mathtt{x}$ and $[\![e]\!]\rho$ for the concrete value of the expression e in the concrete environment ρ. The concretization of a segmented decision tree reduced to a leave is

$$\gamma_t((\!| p |\!)) \triangleq \gamma_\ell(p)$$

and the concretisation of a segmented decision tree rooted at a decision node is

$$\begin{aligned}
&\gamma_t.([\![\mathtt{x}\,\{C\} : t_0 b_1 t_1 \dots b_n t_n]\!]) \triangleq \\
&\quad \{\rho \in \gamma_c(C) \mid \forall i \in [1, n) : [\![b_i]\!]\rho \leqslant [\![b_{i+1}]\!]\rho \wedge \\
&\qquad (n = 0 \vee \rho(\mathtt{x}) < [\![b_1]\!]\rho) \Longrightarrow \rho \in \gamma_t(t_0) \wedge \\
&\qquad \forall i \in [1, n) : ([\![b_i]\!]\rho \leqslant \rho(\mathtt{x}) < [\![b_{i+1}]\!]\rho) \Longrightarrow \rho \in \gamma_t(t_i) \wedge \\
&\qquad (n > 0 \wedge \rho(\mathtt{x}) \geqslant [\![b_n]\!]\rho) \Longrightarrow \rho \in \gamma_t(t_n)\}
\end{aligned}$$

[2] For clarity, some redundancy is sometimes preserved in segmented decision trees while, for brevity, some repetitive information is omitted in fixpoint computations.

We introduce also the notation $\bot_{\mathbb{D}}$ for the decision tree in $\mathbb{T}((\mathbb{D}, <_{\mathbb{D}}), \mathbb{E}, D_c, D_\ell)$ such that each node is of the form $[\![\, \mathtt{x}\, \{\top_C\} : t \,]\!]$ and the only leaf is $(\!|\, \bot_\ell \,|\!)$, where $\top_C$ is the top element of D_c and $\bot_\ell$ is the bottom element of D_ℓ. When $\mathbb{D}$ is clear from the context, we simply write $\bot$.

7.1 Segmented Decision Tree Abstract Functor

The segmented decision tree abstract functor $\mathbb{T}((\mathbb{D}, <_{\mathbb{D}}), \mathbb{E}, D_c, D_\ell)$ is a parameterized abstract domain taking as a parameter a totally ordered set $(\mathbb{D}, <_{\mathbb{D}})$ of decision variables, a set $\mathbb{E}$ of canonical expressions, an ordering abstract domain D_c and a leaf abstract domain D_ℓ for the leaves.

The abstract domain D_ℓ for the leaves is usually the reduced product [6] of several abstract domains, as was the case for binary decision trees in Sect. 4. The list of abstract domains appearing in this reduced product at the leaves is assumed to be an option of the static analyzer constructor. Therefore this option specifies a particular instance of the segmented decision tree abstract functor used to build a particular instance of the static analyzer for that option. The advantage of this modular approach is that the static analyzer can be changed by changing the options, without any re-programming.

The maximal height of the segmented decision trees is a parameter of the static analysis which can therefore be changed before each run of the static analyzer. A variable packing pre-analysis is used to determine which variables $\mathbb{D}$ are chosen to appear in the decision and leave nodes. The number of variables in the decision nodes is bounded by this maximal height. Following [11], the choice of which expressions $b_1, \ldots, b_n \in \mathbb{E}$, $n \geqslant 0$ do appear in decision nodes is made during the static analysis.

7.2 Reduction of an Abstract Property by a Segmented Decision Tree

Given a segmented decision tree t and an abstract property $p \in D$ of the variables in abstract domain $\langle D, \sqsubseteq, \bot, \sqcup, \sqcap\rangle$ with concretization γ, $t \sqcap_D p$ is the abstraction of the conjunction $\gamma_t(t) \cap \gamma(p)$ in the abstract domain D. It is the intersection of p with the join of the abstract properties obtained along paths of t feasible for p.

Example 6. In *Ex. 5*, the hypotheses that B_1 is *true* and $X < M$ imply that $Y \in [-1, 10] \sqcup [0, 0] = [-1, 10]$. The implied condition collects information along the path and at the leaves and is therefore $B_1 \wedge (X < M) \wedge (1 < J = K + 1 < M \leqslant N) \wedge Y \in [-1, 10]$. □

The operation $t \sqcap_D p$ is used in the definition of the reduced product of the abstract domain $\langle D, \sqsubseteq\rangle$ by the abstract domain of segmented decision trees.

A path of t is feasible for p if it corresponds, at each level in the tree, to a segment of the node which, according to p and the conjunction of side conditions collected from the root to this node, is not empty. For each decision variable, the conjunction of the hypothesis p and the collected side conditions is used to

determine to which segments the value of the variable does belong. The information available on this variable is thus the join of the information available at the leaves for each segment plus the fact that the variable value is between the extreme bounds of these segments along this path. More formally ($\sqsubseteq$ is the abstract implication, $(\,true\ ?\ a : b\,) \triangleq a$, $(\,false\ ?\ a : b\,) \triangleq b$ is the conditional, $D(p)$ is the best approximation of $p \in D_c \cup D_\ell$ in D (in absence of best abstraction, an over-approximation must be used)).

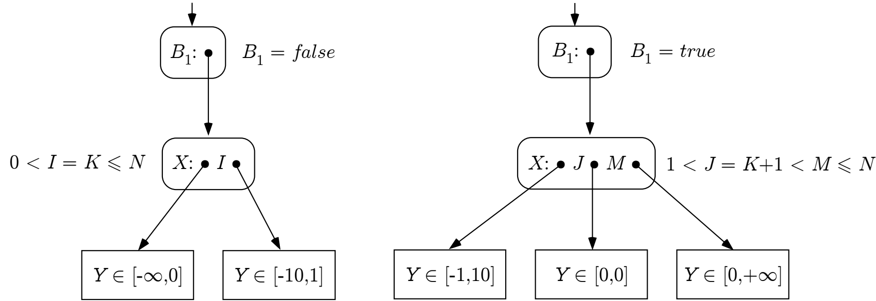
$$\begin{aligned}
\bot_{\mathbb{D}} \sqcap_D p &\triangleq \bot \\
t \sqcap_D \bot &\triangleq \bot \\
(\!|\, p' \,|\!) \quad \sqcap_D p &\triangleq D(p') \sqcap p \\
[\![\, \mathtt{x}\,\{C\} : t_0 b_1 t_1 \ldots b_n t_n \,]\!] \quad \sqcap_D p &\triangleq t_0 \sqcap_D (p \sqcap D(C) \sqcap (\, n = 0 \ ?\ \top : D(\mathtt{x} < b_1))\,) \\
&\quad \sqcup \bigsqcup_{i=1}^{n-1} t_i \sqcap_D (p \sqcap D(C) \sqcap D(b_i \leqslant \mathtt{x} < b_{i+1})) \\
&\quad \sqcup(\, n > 0\ ?\, t_n \sqcap_D (p \sqcap D(C) \sqcap D(\mathtt{x} \geqslant b_n)) : \bot)
\end{aligned}$$

Observe that $p \sqsubseteq q$ implies $\gamma(p) \subseteq \gamma(q)$ whereas $p \not\sqsubseteq q$ does not implies that $\gamma(p) \not\subseteq \gamma(q)$ whereas the sufficient condition $p \sqcap q = \bot$ implies $\gamma(p) \cap \gamma(q) = \emptyset$ and so $\gamma(p) \not\subseteq \gamma(q)$.

7.3 Reduction of a Segmented Decision Tree by an Abstract Property, Tests

In a test, all paths that are feasible in the segmented decision tree are preserved while all the paths that, for sure, can never be followed according to the tested condition are disregarded.

Example 7. Assume that in *Ex. 5*, the test is on B_1. On the *true* branch of the test, the *false* subtree is disregarded while on the *false* branch of the test the *true* subtree is disregarded. We get:



□

Formally, we define the restriction $t \sqcap_t p$ of the segmented decision tree t by condition $p \in D$ as

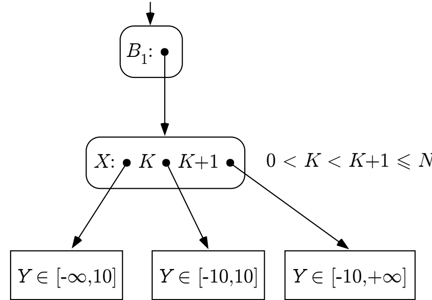
$$\begin{array}{l}
\bot_{\mathbb{D}} \sqcap_t p \triangleq \bot_{\mathbb{D}} \\
t \sqcap_t \bot \triangleq \bot_{\mathbb{D}} \\
\quad (\!| p' |\!) \quad \sqcap_t p \triangleq \quad (\!| p' \sqcap_{D_\ell} D_\ell(p) |\!) \\
\quad [\![\, \mathtt{x} \, \{C\} : \, t_0 b_1 t_1 \ldots b_n t_n \,]\!] \quad \sqcap_t p \triangleq \\
\quad \text{let} \quad t'_0 \triangleq t_0 \sqcap_t (p \sqcap D(C) \sqcap (\, n = 0 \, ? \, \top : D(\mathtt{x} < b_1) \,)) : \\
\quad \text{and for } i = 1, \ldots, n-1 : \; t'_i \triangleq t_i \sqcap_t (p \sqcap D(C) \sqcap D(b_i \leqslant \mathtt{x} < b_{i+1})), \\
\quad \text{and if } n > 0, \; t'_n \triangleq t_n \sqcap_t (p \sqcap D(C) \sqcap D(\mathtt{x} \geqslant b_n)) \\
\quad \text{in } (\, (\forall i \in [0, n] : t'_i = \bot_{\mathbb{D}}) \; ? \; \bot_{\mathbb{D}} : \\
\qquad \text{let } l = \min\{i \in [0, n] \mid t'_i \neq \bot_{\mathbb{D}}\} \text{ and } m = \max\{i \in [0, n] \mid t'_i \neq \bot_{\mathbb{D}}\} \text{ in} \\
\qquad\quad [\![\, \mathtt{x} \, \{\mathrm{relax}_C \, (C, p)\} : L'_l b_{l+1} t'_{l+1} \ldots b_m t'_m \,]\!] \quad)
\end{array}$$

Note that the information of p that changes the ordering is kept global. On the contrary, it is possible to relax the incremental information on pre-order at each node: each constraint implied by both C and p can safely be removed from C, leading to a more compact representation. The choice of computing that relaxation depends on the domain for C. This choice is noted $\mathrm{relax}_C \, (C, p)$ in the algorithm. In addition, we can perform another optimization when a node is left with only one child. In that case, we can join the pre-order of the node and its child and put top as the incremental pre-order for the child.

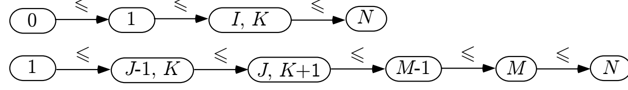
7.4 Segments Unification, Tree Merges and Binary Operations

When performing a binary operation on two decision trees (joins, widening, ordering. . .), we have to go through the two trees at the same time. Because the variables are ordered, for each pair of subtree during the traversal, the root is on the same decision variable, but the segments may have no bound in common. So, in order to push the binary operation to subtrees, we need to find a common refinement for the two segments (as given by the refinement algorithm in [11]), which we call segments unification.

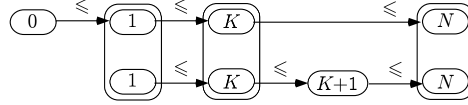
Example 8. Consider the random assignment $B_1 =?$ to the boolean variable B_1 in the context of *Ex. 5.* The subtrees of the two segments of B_1 must be merged, as follows



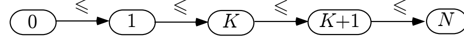
The pre-orders on the decision variable X in *Ex. 5* involve the bounds of the decision variable and the equivalence classes of the expressions appearing in the segmentation.



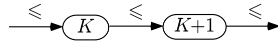
The union of these pre-orders eliminates the variables I, J, and M since they are not comparable in both pre-orders. For example the first pre-order may correspond to a program context where $I = K$ but this might not hold in the program context corresponding to the second pre-order. However although K and N might have different values in these two program contexts, the relation $K \leqslant N$ is valid in both program contexts and so is preserved in the union of the pre-orders.



This union contains only one maximal chain



which yields a relation between (classes of equal) expressions which is valid in both pre-orders and can therefore be attached to the merged node for X. The segmentation for X in the merged tree is the subchain obtained by considering classes of expressions with representatives appearing in either of the original segmentations (that is $K = I$ and $K + 1 = J$ while 0, 1 and N did not appear).



- The subtree in the refined first segment $X < K$ is the merge of the subtrees of the corresponding segment $Y \in [-\infty, 0]$ on the left ($X < I = K$) and $Y \in [-1, 10]$ on the right ($X < J = K + 1$), that is $Y \in [-\infty, 0] \sqcup [-1, 10] = [-\infty, 10]$.
- The subtree in the refined second segment $K \leqslant X < K + 1$ is the merge of the subtrees of the corresponding segments on the left ($Y \in [-10, 1]$ when $X \geqslant I = K$) and on the right ($Y \in [-1, 10]$ when $X < J = K + 1$), that is $Y \in [-10, 1] \sqcup [-1, 10] = [-10, 10]$.
- The subtree in the refined third and last segment $X \geqslant K + 1$ is the merge of the subtrees of the corresponding segments on the left ($Y \in [-10, 1]$ for $X \geqslant I = K$) and on the right ($Y \in [0, 0] \sqcup [0, +\infty]$ for $J = K + 1 \leqslant X < M$ or $M \leqslant X$), that is $Y \in [-10, 1] \sqcup [0, 0] \sqcup [0, +\infty] = [-10, +\infty]$). □

In contrast with simple array segmentation, we may have richer ordering informations, and it will be useful to provide precise segments unification. Given

two nodes $[\![\, \mathtt{x}\, \{C^0\} : t^0_0 b^0_1 \ldots b^0_{n^0} t^0_{n^0} \,]\!]$ and $[\![\, \mathtt{x}\, \{C^1\} : t^1_0 b^1_1 \ldots b^1_{n^1} t^1_{n^1} \,]\!]$, assuming that we collected all the preordering informations in C^0 and C^1, we must compute two nodes $[\![\, \mathtt{x}\, \{C^0\} : t'^0_0 b_1 \ldots b_n t'^0_n \,]\!]$ and $[\![\, \mathtt{x}\, \{C^1\} : t'^1_0 b_1 \ldots b_n t'^1_n \,]\!]$ which do share the same bounds and are sound (and precise) over-approximations of the input nodes. For these nodes to be valid, the bounds $b_1 \ldots b_n$ must respect the ordering in both C^0 and C^1. So, the first step is to compute $C^0 \cup_C C^1$. Then the resulting nodes will be more precise if the new bounds can all be compared to the previous bounds. That is, for all $k \in \{0, 1\}$, for all $i \le n^k$ and all $j \le n$, either $\forall \rho \in \gamma_c(C^k)[\![b^k_i]\!]\rho \le [\![b_j]\!]\rho$ or $\forall \rho \in \gamma_c(C^k)[\![b^k_i]\!]\rho \ge [\![b_j]\!]\rho$. Then solving the segment unification problem consists in finding a chain in the pre-order defined by $C^0 \cup_C C^1$ such that all elements of the chain respect that property. In general there is no best chain, but the longer the chain the better, and it is certainly best to maximize the number of bounds b^k_i such that there is a b_j such that $\forall \rho \in \gamma_c(C^k)[\![b^k_i]\!]\rho = [\![b_j]\!]\rho$, meaning that a maximum number of bounds are preserved.

The algorithm to find the bounds $b_1 \ldots b_n$ will thus starts by building a graph whose vertices are elements of each segments and the expressions in E that are equal to a bound in each segment according to the C^k's (including extrema for the decision variable, if any). Then on that graph, we merge the strongly connected components. Each vertex of this graph can be colored by a couple of bounds (b^0_i, b^1_j) or by one bound of a segment. Then we find the path in the graph with maximal number of colors. A couple of optimizations can be used to compute that path, and any path is a correct, although maybe imprecise, answer, so we can also stop that algorithm at any point based on a time limit.

Once we have computed the bounds $b_1 \ldots b_n$, we compute the new subtrees

$$\begin{aligned} & [\![\, \mathtt{x}\, \{C^0\} : t^0_0 b^0_1 \ldots b^0_{n^0} t^0_{n^0} \bowtie_{C^0} b_1 \ldots b_n \,]\!] \\ \text{and} \quad & [\![\, \mathtt{x}\, \{C^1\} : t^1_0 b^1_1 \ldots b^1_{n^1} t^1_{n^1} \bowtie_{C^1} b_1 \ldots b_n \,]\!] \end{aligned}$$

where:

$t_0 a_1 t_1 \ldots a_m t_m \bowtie_C b_1 \ldots b_n =$

- if $m = 0$ then
  - if one of the b_i is such that $\mathtt{x} < b_i$ in C, let l be the smallest such i. The result is $t_0 b_1 \ldots b_{l-1} t_0 b_l \bot b_{l+1} \ldots b_m \bot$
  - else $t_0 b_1 \ldots b_n t_0$
- else if $n = 0$ then $t_0 \cup \ldots \cup t_m$
- else if $b_1 = a_1$ in C, we must look at b_2, if any, in case of segment creation:
  - if $n > 1$ and $b_2 = b_1$ in C, then we create a segment. The value of that segment depends on the binary operation (as in [11]). If the operation is a join, a widening or the segment is the first argument of inclusion testing then the value is $\bot$. If the operation is a meet, or a narrowing, or the second argument of an inclusion testing then the value is $\top$. Let us call $\mathbb{I}$ that value. the result is then $t_0 b_1 (\mathbb{I} a_1 t_1 \ldots a_m t_m \bowtie_C b_2 \ldots b_n)$
  - else the result is $t_0 b_1 (t_1 a_2 t_2 \ldots a_m t_m \bowtie_C b_2 \ldots b_n)$
- else if $a_1 \le b_1$ in C then

  – if $m > 1$ and $b_1 \leq a_2$ in C then $(t_0 \cup t_1)b_1t_1a_2 \ldots a_mt_m \bowtie_C b_1 \ldots b_n$
  – else $(t_0 \cup t_1)a_2t_2 \ldots a_mt_m \bowtie_C b_1 \ldots b_n$

— else $(b_1 \leq a_1)$,
  – if $b_1 \leq \mathtt{x}$ in C then $\bot b_1(t_0a_1t_1 \ldots a_mt_m \bowtie_C b_2 \ldots b_n)$
  – else $t_0b_1(t_0a_1t_1 \ldots a_mt_m \bowtie_C b_2 \ldots b_n)$

Once we have computed two new trees which agree on the bounds, we can perform the binary operation. The binary operation must be carried on the incremental pre-orders and on each pair of subtrees recursively, until reaching the leaves where we can compute the binary operation on D_ℓ.

Join. There is a special case when joining two trees: in case we create new segments, there is an opportunity to add incremental pre-order informations. If we merge $[\![\, \mathtt{x}\,\{C\} : \ldots]\!]$ in a context of pre-order information C_1, and $\bot$ in the context of C_2, then the result is $[\![\, \mathtt{x}\,\{C \sqcap_c (\mathrm{relax}_C\,(C_1, C_2))\} : \ldots]\!]$ because we know that on that segment only the pre-order in C_1 is true. For that mechanism to be precise, we need to keep track of the pair of possible pre-orders during the computation of the joins.

Widening. Because segments are split, their number in a decision node can grow indefinitely so that the segmented decision tree can explode in breadth. This must be prevented by a segment widening. We are in a situation where one abstract domain (the segments of a node) is a basis for another one. We can use a mechanism similar to [22] and widen on the segmentations, then on that common segmentation apply the widening child by child.

A first possibility for the segmentation widening is the segments unification of [11] which is based on the use of the common expressions in the two segmentations. In that way, the number of expressions that appear in segments can only decrease. We can achieve the same property by keeping all expressions that can occur as bounds of the first argument of the widening and only those expressions. This is easily implemented by keeping only those expressions in the graph where we look for a best chain of expressions.

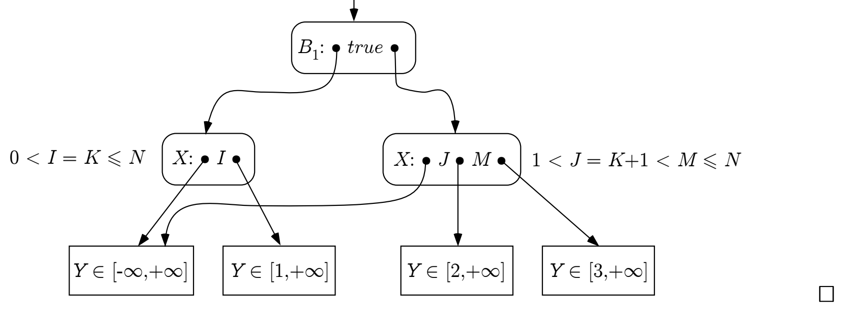
7.5 Assignments

As variables may occur in leaves, expression bounds or as decision variables, we have to consider all cases, keeping in mind that those cases are not exclusive.

Assignments to leaf variables. An assignment to a variable appearing in the leaf nodes only will determine the feasible paths to the leaves where it appears and perform the assignments in each of these leaves (in the abstract domain of the leaves).

Example 9. Assuming in *Ex. 5* that nothing is known on the upper bound of I, J, K, M, and N in the variable environment, the assignment $Y = X$ will determine that either $\neg B_1$ in which case if $X < I$ then else $X \geqslant I > 0$ so

$Y \in [1, +\infty]$ or B_1 holds and so either $X < J$ in which case $Y \in [-\infty, +\infty]$, or $1 < J \leqslant X$ so $Y \in [2, +\infty]$, or else $1 < J < M \leqslant X$ and so $Y \in [2, +\infty]$. We get



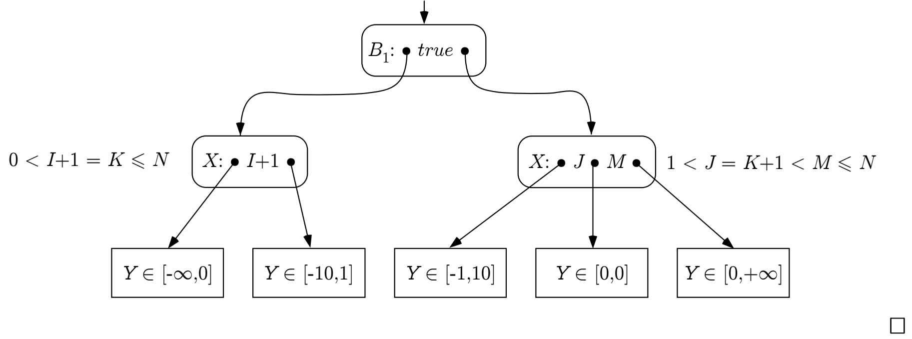
□

In general the assignment of an expression to a variable involves some conditions (such as absence of overflow, positiveness, non-nullity, etc) that have to be taken into account by pruning the tree as in Sect. 7.3. In case where we have to do such pruning, we can follow the same algorithm, but performing the assignment at the leaves in addition to imposing the test.

Assignments to segment bound variables. An assignment to a variable appearing in segment bounds may be invertible, in which case segments which were based on the old value of the variable can be expressed based on the new value, or not invertible, in which case it is not possible to keep the segments bounds when they are only expressible in terms of the old values of the assigned variable.

More precisely, if the assignment can be expressed as $b = f(b)$ and f invertible, we can replace the variable b by $f^{-1}(b)$ in each expression appearing in bounds of the decision tree, and that encodes the same property. To complete the assignment, we must also carry it to incremental pre-orders at each node.

Example 10. Consider the assignment $I = I - 1$ in the context of *Ex. 5*. After this assignment the old value I_o of I (to which *Ex. 5* is referring to) can be expressed in terms of the new value I_n as $I_n = I_o - 1$ so $I_o = I_n + 1$ by inversion. So, we get the post-condition of the assignment $I = I - 1$ by replacing I by $I + 1$ in the segmented decision tree of *Ex. 5*.



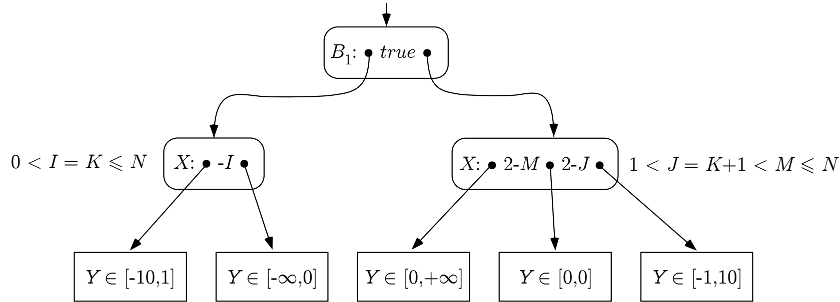
□

In case of non-invertible assignment to a variable b appearing in a bound, we look at all bounds were b appears. If at that bound the pre-order information can provide another expression that is known to be equal to the bound but that does not contain b, we can replace the bound by that expression. Otherwise, we drop the bound from the segmentation and merge the two consecutive subtrees that were separated by that bound. As for the tests, if that results in only one child for a node, we can push up the incremental pre-order information of that child.

In addition, the assignment must be carried also in the incremental pre-orders.

Assignments to decision variables. For an invertible assignment $X = f(X)$ to a decision variable X where X_o (resp. X_n) is the value of X before (resp. after) the assignment, we have $X_n = f(X_o)$ such that $X_o = f^{-1}(X_n)$. The segment conditions $b \leqslant X_o$ are transformed into $b \leqslant f^{-1}(X_n)$ that is $f(b) \leqslant X_n$ when f is increasing and $f(b) \geqslant X_n$ when f is decreasing. Similarly $X_o < b$ are transformed into $f^{-1}(X_n) < b$ that is $X_n < f(b)$ when f is strictly increasing and $x_n > f(b)$ when f is strictly decreasing. If f also depends on other decision variables, their abstract value must be evaluated along the path to the nodes for X.

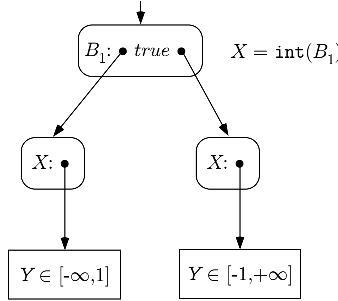
Example 11. Consider the invertible assignment is $X = \texttt{int}(B_1) - X$ in *Ex. 5* where $\texttt{int}(\mathit{false}) = 0$ and $\texttt{int}(\mathit{true}) = 1$ that is $X = -X$ when B_1 is *false* and $X = 1 - X$ when B_1 is *true* , which are both invertible assignments.



□

In case of non-invertible assignment, we cannot keep the segments related to the assigned variable. So we merge the children of that variable. In case where the assignment `x=e` can be represented exactly in the pre-order domain, then we are as precise as possible. Otherwise, it is still possible to add some information in the tree. For example if the expression is a monotone function over another variable `y` smaller than `x` (for $<_{\mathbb{D}}$) then we can store the inequalities implied by the segmentation over `y` into the incremental pre-orders associated with `y`. If the variable is greater than `x`, we can do the same if all the nodes on `y` share a bound.

Example 12. Consider the non-invertible assignment is $X = \texttt{int}(B_1)$. The post-condition is preserved while selectively merging the children. Assuming $\texttt{int}(b)$ to be a canonical integer expression for canonical Boolean expressions b, we get:



□

8 Abstracting Functions (and Array Contents)

Binary Decision Diagrams were originaly developped to represent boolean functions [1]. In the same way, segmented decision trees can be used to approximate functions over totaly ordered domains: we make decisions for each parameter of the function, and the leaves of the tree represent the possible values of the function for that constraint on the parameters.

Example 13. The function $\sin x$, $x \in [0, 2\pi]$ could be approximated by the segmented decision tree $[\![\, x \,\{0 \leqslant x \leqslant 2\pi\} : (\!|\, \sin x : [0,1] \,|\!) \;\pi\; (\!|\, \sin x : [-1,0] \,|\!)]\!]$. □

Formaly, a function $f(x_0, \ldots, x_n)$ can be seen as a set of vectors of size $n + 1$ of the form $< v_0, \ldots, v_n, f(v_0, \ldots, v_n) >$. Then a property over functions is a set of sets of vectors. The first abstraction we perform is to go back to sets of vectors, by taking the union of the sets, then we are in a setting where we can use segmented decision trees directly, with decision variables the first n variables ordered by the order on parameters of the function. Such abstraction could be very powerful to summarize functions and perform modular analyzes.

Multi-dimensional arrays can be seen as functions from index values to array content. So we can use the same combination of abstractions and obtain precise representations. Because arrays don't have formal parameters, we just need a convention to name to variables which will correspond to the array dimensions and to the array content. One possibility is to subscript the array name with the number of the dimension for the indexes and with v for the content.

One more important difference between functions and arrays is the assignment. It is easily implemented as an assignment to the content variable under the appropriate condition for the dimension variables. May-assign can arize when such conditions cannot be represented exactly in the expressions allowed as bounds, but in general this mechanism will be very precise.

If we specialize that abstraction to arrays of dimension 1, we have an abstraction that is equivalent to [11] where the leaves consisted of abstractions of couples index-content.

9 Examples

9.1 Conditional Computation

```
          int x1, x2, y, z;
          struct s;
/*  0: */  x1 = 0;
          y = z = 1;
          s = INIT;
/*  1: */  while /* 2: */( z < 100 ) {
/*  3: */    if (x1 < y) s = INIT; /* 4: */
             else { /* 5: */
                if (x2 > y) s = SPECIAL; /* 6: */
                else s = computation(s, x2); /* 7: */
/*  8: */    }
/*  9: */    z++;
/* 10: */    if (?) y++; /*11: */
/* 12: */    if (?) x1++;/*13: */
/* 14: */  }
/* 15: */  if (x1>y && x2>y) /* 16: */ assert(s==SPECIAL);
```

In that abstract program, the structure `s` is the output and the variable `x2` is the input. On some variation, `x2` may change at each loop iteration but that does not change the invariants here, so we just consider that its value is fixed.

A pre-analysis can fix the decision variables to be `x1` and `x2` as they are in guards to compute values and then in a guard to test those values. The structure `s` will be at the leaves (the result of `computation` is abstracted by `COMP`).

The fixpoint iteration with widening will go as follows (we write $\llbracket\, \mathtt{v} : \ldots \rrbracket$ for $\llbracket\, \mathtt{v}\, \{true\} : \ldots \rrbracket$):

1-9: $\llbracket\, x1\, \{x1 = 0,\, y = z = 1\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket \rrbracket$ $\quad$ ⦃Initialization with input $x2$⦄

14: $\llbracket\, x1\, \{x1 \in [0, 1],\, y \in [1, 2],\, z = 2\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket \rrbracket$

2: $\llbracket\, x1\, \{0 \le x1 \le y \le z,\, x1 < z,\, 0 < y\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket \rrbracket$

3: $\llbracket\, x1\, \{0 \le x1 \le y \le z,\, x1 < z,\, 0 < y\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket \rrbracket$

4: $\llbracket\, x1\, \{0 \le x1 < y \le z < 100\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket \rrbracket$

5: $\llbracket\, x1\, \{0 < y = x1 < z < 100\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket \rrbracket$

6: $\llbracket\, x1\, \{0 < y = x1 < z < 100,\, y < x2\} : \llbracket\, x2 : \llparenthesis\, \mathtt{SPECIAL}\, \rrparenthesis \rrbracket \rrbracket$

7: $\llbracket\, x1\, \{0 < y = x1 < z < 100,\, x2 \le y\} : \llbracket\, x2 : \llparenthesis\, \mathtt{COMP}\, \rrparenthesis \rrbracket \rrbracket$

8: $\llbracket\, x1\, \{0 < y = x1 < z < 100\} : \llbracket\, x2 : \llparenthesis\, \mathtt{COMP}\, \rrparenthesis \rrbracket\ y + 1\ \llparenthesis\, \mathtt{SPECIAL}\, \rrparenthesis \rrbracket$

9: $\llbracket\, x1\, \{0 \le x1 \le y \le z < 100,\, x1 < z,\, 0 < y\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket\ y\ \llbracket\, x2 : \llparenthesis\, \mathtt{COMP}\, \rrparenthesis \rrbracket$
$\quad y + 1\ \llparenthesis\, \mathtt{SPECIAL}\, \rrparenthesis \rrbracket$

10: $\llbracket\, x1\, \{0 \le x1 \le y < z < 101,\, x1 < z - 1,\, 0 < y\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket$
$\quad y\ \llbracket\, x2 : \llparenthesis\, \mathtt{COMP}\, \rrparenthesis \rrbracket\ y + 1\ \llparenthesis\, \mathtt{SPECIAL}\, \rrparenthesis \rrbracket$

11: $\llbracket\, x1\, \{0 \le x1 < y \le z < 101,\, x1 < z - 1,\, 1 < y\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket$
$\quad y - 1\ \llbracket\, x2 : \llparenthesis\, \mathtt{COMP}\, \rrparenthesis \rrbracket\ y\ \llparenthesis\, \mathtt{SPECIAL}\, \rrparenthesis \rrbracket$

12: $\llbracket\, x1\, \{0 \le x1 \le y \le z < 101,\, x1 < z - 1,\, 0 < y\} : \llbracket\, x2 : \llparenthesis\, \mathtt{INIT}\, \rrparenthesis \rrbracket\ y - 1$

$\llbracket x2 : \ (\!|\,\texttt{COMP} \cup \texttt{INIT}\,|\!)\ y\ (\!|\,\texttt{COMP} \cup \texttt{SPECIAL}\,|\!)\ \rrbracket\ y$

$\llbracket x2 : \ (\!|\,\texttt{COMP}\,|\!)\ y\ (\!|\,\texttt{COMP} \cup \texttt{SPECIAL}\,|\!)\ y+1\ (\!|\,\texttt{SPECIAL}\,|\!)\ \rrbracket \rrbracket$

`13:` $\llbracket x1\,\{1 < x1 < z < 101,\ 0 < y \leq z\} : \llbracket x2 : \ (\!|\,\texttt{INIT}\,|\!)\ \rrbracket\ y$

$\llbracket x2 : \ (\!|\,\texttt{COMP} \cup \texttt{INIT}\,|\!)\ y\ (\!|\,\texttt{COMP} \cup \texttt{SPECIAL}\,|\!)\ \rrbracket\ y+1$

$\llbracket x2 : \ (\!|\,\texttt{COMP}\,|\!)\ y\ (\!|\,\texttt{COMP} \cup \texttt{SPECIAL}\,|\!)\ y+1\ (\!|\,\texttt{SPECIAL}\,|\!)\ \rrbracket \rrbracket$

`14:` $\llbracket x1\,\{0 \leq x1 < z < 101,\ 0 < y \leq z\} : \llbracket x2 : \ (\!|\,\texttt{INIT}\,|\!)\ \rrbracket\ y-1$

$\llbracket x2 : \ (\!|\,\texttt{COMP} \cup \texttt{INIT}\,|\!)\ y\ (\!|\,\texttt{COMP} \cup \texttt{SPECIAL} \cup \texttt{INIT}\,|\!)\ \rrbracket\ y$

$\llbracket x2 : \ (\!|\,\texttt{COMP} \cup \texttt{INIT}\,|\!)\ y\ (\!|\,\texttt{COMP} \cup \texttt{SPECIAL}\,|\!)\ \rrbracket\ y+1$

$\llbracket x2 : \ (\!|\,\texttt{COMP}\,|\!)\ y\ (\!|\,\texttt{COMP} \cup \texttt{SPECIAL}\,|\!)\ y+1\ (\!|\,\texttt{SPECIAL}\,|\!)\ \rrbracket \rrbracket$

`2:` $=$ `14:` without $z < 101$

≀as the union of `14:` and `2:` is `14:` here and this is the abstract loop invariant∫

`15:` $=$ `2:`

`16:` $\llbracket x1\,\{0 < y < x1 < z,\ y < x2,\ 99 < z\} : \llbracket x2 : \ (\!|\,\texttt{SPECIAL}\,|\!)\ \rrbracket \rrbracket$

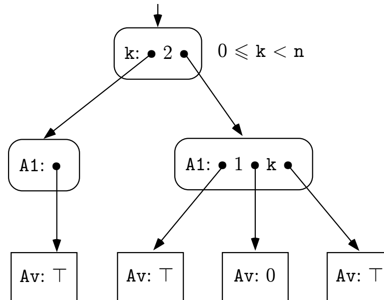
≀The assertion in `16:` is proved correct.∫

9.2 Partial Array Initialization

The program below partially initializes an array `A`.

```
         int n; /* n > 0 */
         int k, A[n];
/* 0: */ k = 0;
/* 1: */  while /* 2: */ (k < n) {
/* 3: */     if (k > 0) {
/* 4: */        A[k] = 0;
/* 5: */      };
/* 6: */      k = k+1;
/* 7: */  };
/* 8: */
```

The ordering abstract domain D_c is assumed to be the octagon abstract domain [20]). Following Sect. 8, an array `A` is abstracted by two fresh variables $\texttt{A1} \in \mathbb{D}$ to segment indices `A1` of array `A`, $\texttt{A1} \in [\texttt{A}.\mathit{low}, \texttt{A}.\mathit{high}]$ and a variable $\texttt{Av} \in \mathbb{D}$ standing for any value of the array in a given segment such that $\texttt{Av} <_{\mathbb{D}} \texttt{A1}$ and `Av` is a leave. For leaves we use constant propagation [15]. The loop invariant found at point `3` is



The fixpoint iteration with widening is the following:

$\texttt{0:}\quad \llbracket\, \mathtt{k}\, \{0 < \mathtt{n}, 0 \leqslant \mathtt{A1} < \mathtt{n}\} : \llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket$ $\quad\wr\,\mathtt{k}$ and $\mathtt{A}$ uninitialized$\,\wr$

$\ell\texttt{:}\quad \bot$ $\quad\wr\,\ell = \mathtt{1}, \ldots, \mathtt{8}$, infimum$\,\wr$

$\texttt{1:,2:,3:,6:}\quad \llbracket\, \mathtt{k}\, \{\mathtt{k} = 0 < \mathtt{n}\} : \llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket$ $\quad\wr\,\texttt{0:}$ where $\mathtt{k} = 0$, $\mathtt{k} < \mathtt{n}$, $\mathtt{k} \leqslant 0\,\wr$

$\texttt{7:}\quad \llbracket\, \mathtt{k}\, \{\mathtt{k} = 1 \leqslant \mathtt{n}\} : \llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket$ $\quad\wr\,\texttt{6:}$ where $\mathtt{k} = \mathtt{k} + 1\,\wr$

$\texttt{2:,3:}\ \llbracket\, \mathtt{k}\, \{0 \leqslant \mathtt{k} \leqslant 1,\ \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket$ $\quad\wr\,$joining $\texttt{1:}$ and $\texttt{7:}$, test $\mathtt{k} < \mathtt{n}\,\wr$

$\texttt{4:}\quad \llbracket\, \mathtt{k}\, \{1 = \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket$ $\quad\wr\,\texttt{3:}$ with $\mathtt{k} > 0\,\wr$

$\texttt{5:}\quad \llbracket\, \mathtt{k}\, \{1 = \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ 2\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket$

$\wr\,\texttt{4:}$ with $\mathtt{A[k]} = 0$ where $\mathtt{k} = 1\,\wr$

$\texttt{6:}\quad \llbracket\, \mathtt{k}\, \{0 \leqslant \mathtt{k} \leqslant 1,\ \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{\mathtt{k} = 0\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 1$
$\llbracket\, \mathtt{A1}\, \{\mathtt{k} = 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ 2\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket$

$\wr\,$joining $\texttt{3:}$ and $\mathtt{k} \leqslant 0$ so $\mathtt{k} = 0$ together with $\texttt{5:}$ where $\mathtt{k} = 1\,\wr$

$\texttt{7:}\quad \llbracket\, \mathtt{k}\, \{1 \leqslant \mathtt{k} \leqslant 2,\ \mathtt{k} \leqslant \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{\mathtt{k} = 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1}\, \{\mathtt{k} = 2\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ 2\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket$ $\quad\wr\,\texttt{6:}$ where $\mathtt{k} = \mathtt{k} + 1\,\wr$

$\texttt{1:} \sqcup_t \texttt{7:}\ \llbracket\, \mathtt{k}\, \{0 \leqslant \mathtt{k} \leqslant 2,\ \mathtt{k} \leqslant \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{0 \leqslant \mathtt{k} \leqslant 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1}\, \{\mathtt{k} = 2\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ 2\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$ $\quad\wr\,$join of $\texttt{1:}$ and $\texttt{7:}\,\wr$

$\texttt{2:,3:}\quad \llbracket\, \mathtt{k}\, \{0 \leqslant \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{0 \leqslant \mathtt{k} \leqslant 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ \mathtt{k}\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$ $\quad\wr\,\texttt{2:}\ \nabla\ (\texttt{1:} \sqcup_t \texttt{7})$ [3], test $\mathtt{k} < \mathtt{n}\,\wr$

$\texttt{4:}\quad \llbracket\, \mathtt{k}\, \{0 < \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{\mathtt{k} = 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ \mathtt{k}\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$ $\quad\wr\,\texttt{3:}$ with $\mathtt{k} > 0\,\wr$

$\texttt{5:}\quad \llbracket\, \mathtt{k}\, \{0 < \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{\mathtt{k} = 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ 2\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ \mathtt{k}+1\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$ $\quad\wr\,\texttt{4:}$ with $\mathtt{A[k]} = 0\,\wr$

$\texttt{6:}\quad \llbracket\, \mathtt{k}\, \{0 \leqslant \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{\mathtt{k} = 0\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 1$
$\llbracket\, \mathtt{A1}\, \{\mathtt{k} = 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ 2\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ \mathtt{k}+1\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$

$\wr\,$joining $\texttt{3:}$ and $\mathtt{k} \leqslant 0$ with $\texttt{5:}\,\wr$

$\texttt{7:}\quad \llbracket\, \mathtt{k}\, \{0 < \mathtt{k} \leqslant \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{\mathtt{k} = 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1}\, \{\mathtt{k} = 2\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ 2\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 3$
$\llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ \mathtt{k}\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$ $\quad\wr\,\texttt{6:}$ where $\mathtt{k} = \mathtt{k} + 1\,\wr$

$\texttt{1:} \sqcup_t \texttt{7:}\ \llbracket\, \mathtt{k}\, \{0 \leqslant \mathtt{k} \leqslant \mathtt{n}\} : \llbracket\, \mathtt{A1}\, \{0 \leqslant \mathtt{k} \leqslant 1\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ \mathtt{k}\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$ $\quad\wr\,$join of $\texttt{1:}$ and $\texttt{7:}\,\wr$

$\texttt{2:,3:}\quad \llbracket\, \mathtt{k}\, \{0 \leqslant \mathtt{k} < \mathtt{n}\} : \llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ \mathtt{k}\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$

$\wr\,\texttt{2:}\ \nabla\ (\texttt{1:} \sqcup_t \texttt{7})$, test $\mathtt{k} < \mathtt{n}$, convergence, $\texttt{3:}$ is the abstract loop invariant$\,\wr$

$\texttt{8:}\quad \llbracket\, \mathtt{k}\, \{0 \leqslant \mathtt{k} = \mathtt{n}, 0 \leqslant \mathtt{A1} < \mathtt{n}\} : \llbracket\, \mathtt{A1} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ \rrbracket\ 2$
$\llbracket\, \mathtt{A1}\, \{0 \leqslant \mathtt{k} < \mathtt{n}\} :\ \llparenthesis\, \mathtt{Av} : \top \,\rrparenthesis\ 1\ \llparenthesis\, \mathtt{Av} : 0 \,\rrparenthesis\ \rrbracket\, \rrbracket\, \rrbracket$

$\wr\,\texttt{2:}$ and $\mathtt{k} \geqslant \mathtt{n}$, program postcondition$\,\wr$

[3] When a new branch is taken in a test within a loop the widening is usually delayed, which we avoid to shorten the example.

Observe that the segmented decision tree automatically discovers a partition of the loop body as given by the condition $\texttt{k} > 0$ while the segmented array partitions the values of the array elements according to variable `k`.

9.3 Multidimentional Arrays

The program below partially initializes a matrix `M`.

```
          int m, n; /* m, n > 0 */
          int i, j, M[m,n];
/*  0: */ i = 0;
/*  1: */ while /* 2: */ (i < m) {
/*  3: */    j = i+1;
/*  4: */    while /* 5: */ (j < n) {
/*  6: */       M[i,j] = 0;
/*  7: */       j = j+1;
/*  8: */    };
/*  9: */    i = i+1;
/* 10: */ };
/* 11: */
```

A global invariant is $0 \leq \texttt{M1} < \texttt{m}$ and $0 \leq \texttt{M2} < \texttt{n}$, so we keep it implicit in the following fixpoint iteration:

0: $\llbracket \texttt{M1} : \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket \rrbracket$ $\quad\wr$program precondition: `i`, `j`, and `A` uninitialized$\int$

ℓ: $\bot$ $\quad\wr \ell = \texttt{1}, \ldots, \texttt{11}$, infimum$\int$

1:,2:,3: $\llbracket \texttt{M1}\,\{\texttt{i} = 0\} : \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket \rrbracket$ $\quad\wr$`0:` with $\texttt{i} = 0$, $\texttt{i} < \texttt{m}$[4]$\int$

4:,5:,6: $\llbracket \texttt{M1}\,\{\texttt{i} = 0,\, \texttt{j} = \texttt{i} + 1 = 1 < \texttt{n}\} : \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket \rrbracket$
$\quad\wr$`3:` with `j = i+1;`, $\texttt{j} < \texttt{n}\int$

7: $\llbracket \texttt{M1}\,\{\texttt{i} = 0,\, \texttt{j} = \texttt{i} + 1 = 1 < \texttt{n}\} :$
$\quad \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis\ \texttt{j}\ \llparenthesis \texttt{Mv} : 0 \rrparenthesis\ \texttt{j} + 1\ \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket\ \texttt{i} + 1\ \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket$
$\rrbracket$ $\quad\wr$`6:` with `M[i,j] = 0;`$\int$

8: $\llbracket \texttt{M1}\,\{\texttt{i} = 0,\, \texttt{j} = \texttt{i} + 2 = 2 \leqslant \texttt{n}\} :$
$\quad \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis\ \texttt{j} - 1\ \llparenthesis \texttt{Mv} : 0 \rrparenthesis\ \texttt{j}\ \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket\ \texttt{i} + 1\ \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket$
$\rrbracket$ $\quad\wr$`7:` with `j = j+1;`$\int$

4: $\sqcup_t$ 8: $\llbracket \texttt{M1}\,\{\texttt{i} = 0,\, \texttt{i} + 1 \leqslant \texttt{j} \leqslant \texttt{i} + 2 \leqslant \texttt{n}\} :$
$\quad \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis\ 1\ \llparenthesis \texttt{Mv} : 0 \rrparenthesis\ \texttt{j}\ \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket\ \texttt{i} + 1\ \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket$
$\rrbracket$ $\quad\wr$join of `4:` and `8:`$\int$

5: $\llbracket \texttt{M1}\,\{\texttt{i} = 0,\, \texttt{i} + 1 \leqslant \texttt{j} \leqslant \texttt{n}\} :$
$\quad \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis\ 1\ \llparenthesis \texttt{Mv} : 0 \rrparenthesis\ \texttt{j}\ \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket\ \texttt{i} + 1\ \llbracket \texttt{M2} : \llparenthesis \texttt{Mv} : \top \rrparenthesis \rrbracket$
$\rrbracket$ $\quad\wr$`5:` ∇ (`4:` $\sqcup_t$ `8:`) [4]$\int$

[4] When a new branch is taken in a test within a loop the widening is usually delayed, which we avoid to shorten the example.

9: $\llbracket \texttt{M1}\,\{\texttt{i}=0,\,\texttt{i}+1\leqslant\texttt{j}=\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \texttt{i}+1$
$\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\,\rrbracket$ ≀5: and $\texttt{j}\geqslant\texttt{n}$∫

10: $\llbracket \texttt{M1}\,\{\texttt{i}=1,\,\texttt{i}\leqslant\texttt{j}=\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\,\rrbracket$
≀9: and `i = i+1;`∫

1: $\sqcup_t$ 10: $\llbracket \texttt{M1}\,\{\texttt{i}=1,\,\texttt{i}\leqslant\texttt{j}=\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\,\rrbracket$
≀join of 1: and 10:∫

2: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\,\rrbracket$
≀2: ∇ (1: $\sqcup_t$ 10:)∫

3: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}<\texttt{m}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\,\rrbracket$
≀2: and `j < n`∫

4:,5:,6: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}<\texttt{m},\,\texttt{j}=\texttt{i}+1<\texttt{n}\}:$
$\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket$
$\rrbracket$ ≀3:, `j = i+1;` and `j < n`∫

7: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}<\texttt{m},\,\texttt{j}=\texttt{i}+1<\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}$
$\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{j}\ (\!|\texttt{Mv}:0|\!)\ \texttt{j}+1\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\ \texttt{i}+1\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket$
$\rrbracket$ ≀6: and `M[i, j] = 0;`∫

8: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}<\texttt{m},\,\texttt{j}=\texttt{i}+2\leqslant\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \texttt{n}\rrbracket\ \texttt{i}$
$\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{j}-1\ (\!|\texttt{Mv}:0|\!)\ \texttt{j}\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\ \texttt{i}+1\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket$
$\rrbracket$ ≀7: with `j = j+1;`∫

4: $\sqcup_t$ 8: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}<\texttt{m},\,\texttt{i}+1\leqslant\texttt{j}\leqslant\texttt{i}+2\leqslant\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}$
$\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{i}+1\ (\!|\texttt{Mv}:0|\!)\ \texttt{j}\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\ \texttt{i}+1\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket$
$\rrbracket$ ≀join of 4: and 8:∫

5: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}<\texttt{m},\,\texttt{i}+1\leqslant\texttt{j}\leqslant\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}$
$\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{i}+1\ (\!|\texttt{Mv}:0|\!)\ \texttt{j}\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\ \texttt{i}+1\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket$
$\rrbracket$ ≀5: ∇ (4: $\sqcup_t$ 8:)∫

9: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}<\texttt{m},\,\texttt{i}+1\leqslant\texttt{j}=\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}$
$\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{i}+1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}+1\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket$
$\rrbracket$ ≀5: and $\texttt{j}\geqslant\texttt{n}$∫

10: $\llbracket \texttt{M1}\,\{0<\texttt{i}\leqslant\texttt{m},\,\texttt{i}\leqslant\texttt{j}=\texttt{n}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ 1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}-1$
$\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{i}\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket$
$\rrbracket$ ≀9: and `i = i+1;`∫

1: $\sqcup_t$ 10: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}\leqslant\texttt{m}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{M1}+1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\,\rrbracket$
≀join of 1: and 10:(segments unification yields $1\leq\texttt{M1}+1\leq i$ for subtree merges)∫

2: $\llbracket \texttt{M1}\,\{0\leqslant\texttt{i}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{M1}+1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\ \texttt{i}\ \llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \rrbracket\,\rrbracket$
≀2: $\sqsubseteq$ (1: $\sqcup_t$ 10:), stabilization at a fixpoint∫

11: $\llbracket \texttt{M1}\,\{0<\texttt{m}=\texttt{i}\}:\llbracket \texttt{M2}:\ (\!|\texttt{Mv}:\top|\!)\ \texttt{M1}+1\ (\!|\texttt{Mv}:0|\!)\ \rrbracket\,\rrbracket$
≀2: and $\texttt{i}\geqslant\texttt{m}$, program postcondition.∫

10 Conclusion

Many static analyses are very impressive on small examples but fail to scale up. The problem mainly originates from the explosion of possibles cases in handling disjunctions. Mastering the exponential growth is the key to scalability, while enabling weak forms of disjunction is essential to the precision which is necessary to avoid false alarms. Based on two abstract domain functors that have shown experimentally to scale up, we have proposed a new combination which expressivity is better than each of them taken separately and which complexity can be mastered by imposing both static restrictions (like maximal depth or variable packing) and dynamic restrictions (by widening to control the breath of the tree).

References

1. Bryant, R.E.: Graph based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35, 677–691 (1986)
2. Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, March 21 (1978) (in French)
3. Cousot, P.: Verification by abstract interpretation. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 243–268. Springer, Heidelberg (2004)
4. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. 2nd Int. Symp. on Programming, pp. 106–130. Dunod, Paris (1976)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th POPL, Los Angeles, pp. 238–252. ACM Press, New York (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6th POPL, San Antonio, pp. 269–282. ACM Press, New York (1979)
7. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)
8. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
9. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Proc. of the Seventh ACM Conf. on Functional Programming Languages and Computer Architecture, La Jolla, June 25-28, pp. 170–181. ACM Press, New York (1995)
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: The Astrée static analyzer, `www.astree.ens.fr`, `www.absint.com/astree/`
11. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation abstract domain function for the fully automatic and scalable inference of array properties. Research report, Microsoft Research, Redmond (September 2009)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th POPL, Tucson, pp. 84–97. ACM Press, New York (1978)
13. Graf, S., Saïdi, H.: Verifying invariants using theorem proving. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 196–207. Springer, Heidelberg (1996)

14. Granger, P.: Static analysis of arithmetical congruences. Int. J. Comput. Math. 30(3&4), 165–190 (1989)
15. Kildall, G.: A unified approach to global program optimization. In: Ist POPL, Boston, October 1973, pp. 194–206. ACM press, New York (1973)
16. Mauborgne, L.: Analyse statique et domaines abstraits symboliques. Mémoire d'habilitation à diriger les recherches en informatique, Université de Paris Dauphine, February 12 (2007)
17. Mauborgne, L.: Abstract interpretation using TDGs. In: Le Charlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 363–379. Springer, Heidelberg (1994)
18. Mauborgne, L.: Abstract interpretation using typed decision graphs. Science of Computer Programming 31(1), 91–112 (1998)
19. Mauborgne, L.: Binary decision graphs. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 101–116. Springer, Heidelberg (1999)
20. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)
21. Tarski, A.: A lattice theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–310 (1955)
22. Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145. Springer, Heidelberg (1996)