# Tree Schemata and Fair Termination

Laurent Mauborgne

LIENS – DMI, École Normale Supérieure, 45 rue d'Ulm, 75 230 Paris cedex 05, France
Tel: +33 (0) 1 44 32 20 66; Email: Laurent.Mauborgne@ens.fr
WWW home page: http://www.dmi.ens.fr/~mauborgn/

**Abstract.** We present a new representation for possibly infinite sets of possibly infinite trees. This representation makes extensive use of sharing to achieve efficiency. As much as possible, equivalent substructures are stored in the same place. The new representation is based on a first approximation of the sets which has this uniqueness property. This approximation is then refined using powerful representations of possibly infinite relations. The result is a representation which can be used for practical analysis using abstract interpretation techniques. It is more powerful than traditional techniques, and deals well with approximation strategies. We show on a simple example, fair termination, how the expressiveness of the representation can be used to obtain very simple and intuitive analysis.

## 1 Introduction

### 1.1 Trees and Static Analysis

Trees are one of the most widespread structures in computer science. And as such, it is not surprising that sets of trees appear in many areas of static analysis. One of the first practical use of sets of trees in an analysis was presented by Jones and Muchnick in [15], using regular tree grammars to represent sets of trees. The problem was with tree grammars, which are far from ideal, mainly because of the use of set variables. Computing the intersection of two sets, for example, requires the introduction of a quadratic number of variables [1].

In fact, the use of sets of trees have been proposed many times (see [20, 2, 22], and recently all the developments around set based analysis [13]). But all these applications suffer from the same drawbacks, namely the inadequacy of the representation. Practical implementations have been exhibited using tree automata instead of grammars (or sets constraints) [10]. But even tree automata have been introduced at the origin as a theoretical tool for decision problems [23], and they are quite complex to manipulate (see [4, 14, 3] for useful investigations on implementations). Tree automata are also limited in their expressiveness, in that we cannot

express sets of trees with real relationship between subtrees, such as sets of the form $\{f(a^n, b^n, c^n) | n \in \mathbb{N}\}$. They become very complex when we want to add infinite trees, whereas considering infinite behaviors is known to be important in static analysis [21, 24, 6].

When we look closely at those analysis, we see that, due to some lack of expressiveness in the representations, the actual behavior of programs is always approximated in practice. We know a theory to deal smartly with approximations, namely abstract interpretation [7, 8]. And in this framework, we do not need too much from the representations we work with. In particular, there is no need that the sets we can represent be closed by boolean operations, as long as we can *approximate* these operations. What we propose is an entirely new representation for sets of trees —tree schemata—, which is practical and more expressive than traditional techniques (which means finer analysis), taking advantage of the possibilities offered by abstract interpretation.

### 1.2 How to Read the Paper

Tree schemata cannot be extensively described in the frame of one paper. It is the reason why the main ideas leading to these structures have been published in three papers, the present one being the final synthesizing one. The main idea of tree schemata is the use of a first raw approximation, the skeleton, which is then refined by the use of relations. The first approximation is called *skeletons*, and is described in [18]. A short summary of what a skeleton is can be found in this paper in section 2. The relations used in tree schemata may need to relate infinitely many sets, which is why new structures where developed and presented in [16]. These new representations for infinite structures are also described section 3.3.

The rest of the paper is organized as follows: after some basic definitions and the description of skeletons, we show in section 3 how we can enhance them with relations, and still have an incremental representation. The next section describes the expressiveness and some properties of tree schemata, and how they fit in the abstract interpretation framework. Section 5 describes an example of simple analysis exploiting a little bit of the expressiveness of tree schemata.

### 1.3 Basic Definitions and Notations

The trees we consider in this article are possibly infinite trees labeled over a finite set of labels $F$ of fixed arity. A path of the tree is a finite word over $\mathbb{N}$. We write $\prec$ for the prefix relation between paths. The subtree of

a tree $t$ at position $p$ is denoted $t_{[p]}$. A tree is said to be *regular* when it has a finite number of non-isomorphic subtrees. In this case, we can draw it as a finite tree plus some looping arrows.

We will write $\overset{f}{\underset{t_0 \cdots t_{n-1}}{\swarrow \; \searrow}}$ for a generic tree. The label of its root is $f$ of arity $n$, and its children are the $t_i$'s.

We will also consider $n$-ary relations and infinite relations. An $n$-ary relation is defined as a subset of the cartesian product of $n$ sets. The *entry* number $i$ in such a relation corresponds to the $i^{\text{th}}$ position (or set) in the relation. An infinite relation is a subset of an infinite cartesian product.

We will usually use $a$, $b$, $f$, $g$... for labels, $t$, $u$, $v$... for trees, $x$, $y$, $z$ for variables and capitalized letters for sets of trees or structures representing sets of trees.

## 2 Skeletons

Skeletons (see Fig 1) were introduced in [18] as an efficient, yet limited, representation of sets of trees. This representation is based on a canonical representation of infinite regular trees which allows constant time equality testing and very efficient algorithms.

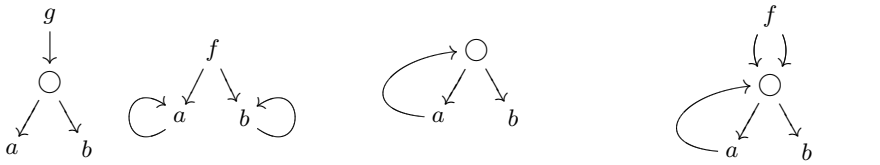



**Fig. 1.** Examples of skeletons

### 2.1 Set Represented by a Skeleton

A skeleton is a regular tree —possibly infinite— with a special label, $\bigcirc$, which stands for a kind of union node.

Let $F$ be a finite set of labels of fixed arity ($\bigcirc \notin F$). A skeleton $\overset{\bigcirc}{\underset{S_0 \cdots S_{n-1}}{\swarrow \; \searrow}}$ will represent the union of the sets represented by the $S_i$, and $\overset{f}{\underset{S_0 \cdots S_{n-1}}{\swarrow \; \searrow}}$ will represent the set of trees starting by an $f$ and such that

its child number $i$ is in the set represented by $S_i$. This definition of the set represented by a skeleton is recursive. In fact, it defines a fixpoint equation. We have two natural ways of interpreting this definition: either we choose the least fixpoint (for set inclusion) or the greatest fixpoint. In the least fixpoint interpretation, a skeleton represents any finite tree that can be formed from it. In the greatest fixpoint interpretation, we add also the infinite tree. As we want the skeletons to be a first approximation to be refined, we choose the *greatest fixpoint*.

### 2.2 Uniqueness of the Representation

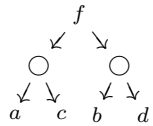
In order to have an efficient and compact representation, skeletons are unique representations of sets of trees. It means that if two sets of trees are equal, they will be stored in the same memory location, making reuse of intermediate results very easy.

In order to achieve this uniqueness, as skeletons are infinite regular trees, we use a representation with this property for infinite regular trees [18]. But we don't have a unique representation for sets of trees yet. We need to restrict skeletons to regular trees labeled by $F \cup \{\bigcirc\}$ and even more:

- in a skeleton, no subtree is the empty skeleton[1] unless the skeleton is the empty skeleton,
- a choice node has either 0 or at least two children,
- a choice node cannot be followed by a choice node,
- each subtree of a choice node starts with a different label. In this way, the choices in the interpretation of a skeleton are deterministic. As a consequence, in addition to common subtrees, we also share common prefixes of the trees, for a greater efficiency.

With these restrictions, skeletons have the uniqueness property, and they are indeed easy to store and manipulate. But the last two rules imply that not every set of trees can be represented by a skeleton. The limitation is that we cannot have any kind of relation between two sets of brother subtree. For example, in the set $\left\{ \begin{smallmatrix} f & & f \\ \swarrow\ \searrow & , & \swarrow\ \searrow \\ a\quad b & & c\quad d \end{smallmatrix} \right\}$, the presence of the subtree $b$ is related to the right subtree $a$, but with a skeleton, the best we can

[1] The empty skeleton is the tree $\bigcirc$, which is a choice with no child.

do is 

. If we did not have infinite trees, the expressive power would be the same as top down deterministic tree automata.

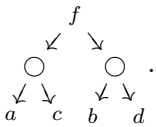
## 3 Links

### 3.1 Choice Space of a Skeleton

Skeletons can be used to give a first upper approximation of the sets we want to represent. Then, we can enrich the skeletons to represent finer sets of trees. A first step towards the understanding of what that means is to define what is the set of possible restrictions we can impose on a skeleton.

The only places in the skeletons where we have any possibility of restriction are choice nodes. Let us consider a choice node with $n$ children. The restrictions we can make are on some of the choices of this node, forbidding for example the second child. So the choice space of a choice node will be the set $\{0, 1, \ldots, n-1\}$. Now, let $S$ be a skeleton. We can make such a restriction for every path of $S$ leading to a choice, and each such restriction can depend on the others.

Thus, the choice space of a skeleton is the cartesian product of all the choice spaces of its choice nodes. Indeed, it gives a new vision of skeletons: we can now see them as a function from their choice space to trees. Each value (which is a vector) in the choice space corresponds to a commitment of every choice nodes in the skeleton to a particular choice.

*Example 1.* Let $S$ be the skeleton 

. Then the choice space of $S$ is $\{0,1\} \times \{0,1\}$. And if we consider $S$ as a function from its choice space, $S(01) =$ 

.

### 3.2 Links are Relations

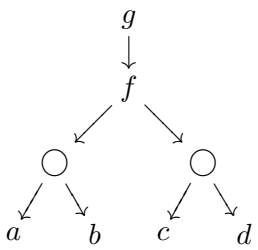
Now we can see clearly what is a restriction of a skeleton: it is a subset of the set of trees it represents which can be defined by choosing a subset of its choice space. And a subset of a cartesian product is merely a relation.

So we have our first definition of tree schemata: a tree schema is a skeleton plus a relation on its choice space.

But this definition raises some problems. First of all, how do we represent the relation? Second, this definition is not *incremental*. A representation is incremental when you do not need to build again the entire representation each time you make a tiny little change in the data. Changes can be made locally, in general. For example, tree automata are not incremental, especially when they are kept minimal, because for each modification of the set, you have to run the minimization algorithm on the whole automaton again. Skeletons are incremental [18]. The advantage of incrementality is clear for the implementation, so we would like to keep tree schemata as incremental as possible. The problem with tree schemata as we have defined them so far is that the relation which binds everything together is global. To change the relation into more local objects, we address two problems: the entries in the relation should not be the paths starting from the root of the tree schema, and the relation should be split if possible. These problems are solved by the notion of links in the tree schema.

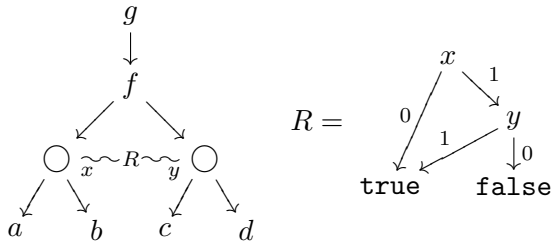
A *link* is a relation with entry names (or variables) [16] plus a function from entry names to sets of choice nodes of the tree schema (formally, a couple (relation, function)). The splitting of the global relation is performed by means of *independent decomposition*. A relation $R$ is independently decomposed in $R_1$ and $R_2$ if: the entries of $R_1$ and $R_2$ partition the entries of $R$, and $R(e)$ is true if and only if $R_1(e_1)$ and $R_2(e_2)$ are true, where $e_i$ is the subvector of $e$ on the entries of $R_i$. The idea is that the global relation is true for a given element of the choice space if and only if it is true on every link. Each choice node is associated with at most one link, and one entry name of that link.

*Example 2.* Consider the following skeleton:




Its choice space is $\{0,1\}_{00} \times \{0,1\}_{01}$ (we use subscripts to denote the entries in the relation, which is the path from the root to the choice node).

A possible restriction would be to consider the set $\left\{ \begin{array}{ccc} g & g & g \\ \downarrow & \downarrow & \downarrow \\ f & f & f \\ \swarrow\ \searrow & \swarrow\ \searrow & \swarrow\ \searrow \\ a\quad c & a\quad d & b\quad d \end{array} \right\}$.
The associated global relation would be $\{0_{00}0_{01}, 0_{00}1_{01}, 1_{00}1_{01}\}$. In order to define the local link, let us call $\square_1$ and $\square_2$ the memory locations of the left and right choice nodes respectively. The local link $l$ would be $(R, x \to \{\square_1\}, y \to \{\square_2\})$, where $R$ is the relation $\{0_x0_y, 0_x1_y, 1_x1_y\}$. In the tree schema, the first choice node would be associated with $(x, l)$ and the second one with $(y, l)$. If we represent the relation by a Binary Decision Diagram (BDD) [5], the tree schema can be depicted this way:




Note that in this example, the letter $R$ appears just for graphical conventions, in tree schemata, links are named just by the representation of their relations and the function from entry names to choice nodes that are associated with them.

### 3.3 Representation of Relations

The last problem concerns the representation of the independent relations. Binary Decision Diagrams [5] having entry names (variables) seem to be a good candidate, as long as the relations are finite! Because the skeleton is an infinite tree, we may have an infinite number of paths leading to choice nodes, and it may be useful to link them together. To achieve this, we need to represent infinite relations, which raise some problems. Those problems have been studied, and a possible solution is presented in [16], which we briefly summarize here. Note that the actual representation of relations is but a *parameter* of tree schemata, and one could choose different representations to change the balance between efficiency and precision, or expressiveness.

**Entry Names** One problem which is common to all representations of infinite relation is that we have an infinite number of entries in the relations, and each of them should be named in order to perform operations

such as restrictions. In BDDs, entry names are the variables, one for each entry in the relation. For infinite relations we can use the notion of *equivalent entries*: two entries $i$ and $j$ are equivalent if for every vector $v$ in the relation, the vector obtained by exchanging its values on the entries $i$ and $j$ is also in the relation. In this case, we show that we can use the same name for $i$ and $j$. This allows the use of a finite number of names, if the relation is regular enough.

**Binary Decision Graphs** In [16], a new class of infinite relations is defined, the set of *omega*-deterministic relations. Intuitively, we can see in relations a finite behavior part, which deals with the prefixes of the vectors, and an infinite behavior. The idea is that for *omega*-deterministic relations, the finite behavior is regular, and at any point in the decision process, there at most one infinite regular behavior.

The representation of such relations is an extension of BDDs: instead of having just DAGs (directed acyclic graphs), we allow cycles in the representation (which are uniquely represented, thanks to the techniques developed in [18]), and we add a special arrow, $\twoheadrightarrow$, which signals the beginning of a new infinite behavior. One can read those graphs as follows (see examples of Fig 2): to accept a vector in the relation, we must follow the decisions in the graph, and count a infinite number of `true`. Each time we encounter a $\twoheadrightarrow$, we reset our count, and each time we encounter a `true`, we start again at the last encountered $\twoheadrightarrow$ (or the beginning of the graph if none was encountered yet). Finite BDDs correspond to the graphs with no cycle and a $\twoheadrightarrow$ before the `true`.

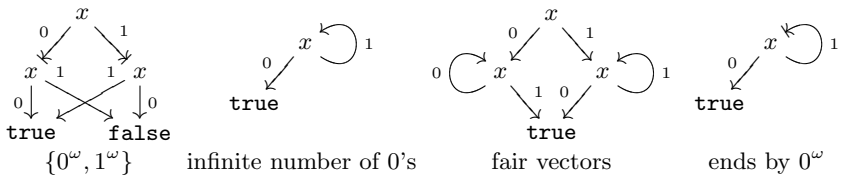



**Fig. 2.** Examples of Binary Decision Graphs

This class of relations is closed by intersection, and has a best (in the sense of relation inclusion) representation property for all boolean operations. Also, the representation is canonical, which gives the constant time equality testing, as with BDDs. It is possible also to represent a bigger

class of infinite relations, the class of *regular* relations, which is closed under all boolean operations, but with far less efficient data structures.

### 3.4 A Pseudo-Decision Procedure

In order to help reading tree schemata, we give a pseudo[2] decision procedure to decide whether a tree is in the set represented by a tree schema. This procedure is performed by going through the tree and the tree schema at the same time. We call $t$ the current subtree, and $T$ the current subtree of the tree schema.

- If $T = \overset{\bigcirc \overset{x}{\sim} R}{\underset{T_0 \cdots T_n}{\swarrow \; \searrow}}$ then
  - if no $T_i$ has the same label as $t$, then the tree is not in the tree schema;
  - otherwise, let $i$ be the index corresponding to the label of $t$. If $R(x = i) = \texttt{false}$ then the tree is not in the tree schema. Else proceed on $T_i$ and $t$, while keeping the fact that $R$ is partially evaluated on $x$ with value $i$.
- $T = \overset{f}{\underset{T_0 \cdots T_{n-1}}{\swarrow \; \searrow}}$ and $t = \overset{g}{\underset{t_0 \cdots t_{n-1}}{\swarrow \; \searrow}}$
  - if $f \neq g$ then the tree is not in the tree schema,
  - else proceed with each $(T_i, t_i)$.

If in this procedure, we need to evaluate a relation on an entry which is already evaluated, we stack a new version of the relation. The procedure is a success if we can go through the entire tree $t$ in this way without failing, and the infinite valuations of relations are accepted. Note that if a relation is not entirely evaluated and there is a possibility of evaluation accepted by the relation, then the process is still a success.

### 3.5 Restrictions on the Links

Just as not every regular tree labeled on $F \cup \{\bigcirc\}$ is a skeleton, not every skeleton with any link is a valid tree schema. There are two main reasons for that: the whole thing must be kept finite (it is a constraint on the representation of the relations only), and we want only one possible representation for the empty set and no infinitely increasing (for the size of the representation) chain of tree schemata representing the same tree.

[2] We call this procedure a "pseudo decision" procedure because the trees and tree schemata being infinite, it cannot end.

Concerning the second constraint, the first thing we need to fix is the skeleton on which the tree schema is based. Because the tree schema represents a subset of the set represented by the skeleton, this skeleton could be any one approximating the set we want to represent. If the set we want to represent admits a best skeleton approximation, it is natural that we choose this skeleton, because the better the first approximation (the skeleton), the more efficient the algorithms. So we choose to put as much information as possible in the skeleton, which corresponds to the arborescent backbone of the set of trees, sharing every possible prefixes and subtrees. In this article, we will restrict tree schemata to such sets of trees, although it is possible to represent sets of trees with no best skeleton approximation, such as $\{a^n b^n c | n \in \mathbb{N}\}$. The reader is referred to [17] for further description.

To restrict the sets we represent to sets with best skeleton approximation, and to keep the skeleton of a tree schema be that best skeleton, we just need to enforce the following two local properties:

*Property 1.* Whatever the link $l$ between two choice nodes $C_1$ and $C_2$, either there is no path from one choice node to the other, or if there is one from $C_1$ to $C_2$, then the choice leading to that path from $C_1$ does not restrict the choices in $C_2$.

*Property 2.* Whatever the link $l$ in a tree schema, the relation of the link is *full*, that is for every entry in the relation and for every possible value at that entry, there is always a vector in the relation with that value on that entry.

A tree schema respecting those properties is said to be valid. In the sequel, we will only consider valid tree schemata.

**Corollary 1.** *Whatever the valid tree schema $T$ based on the skeleton $S$, $S$ is the best (for set inclusion) skeleton approximation for the set represented by $T$.*

*Proof.* Suppose there is a skeleton $S'$ such that $S' \neq S$ and the set represented by $S'$ is included in the set represented by $S$, but still contains the set represented by $T$. It means that there is a path $p$ in $S$ such that $S_{[p]}$ is a choice node and there is a choice $i$ which is not possible in $S'$. The choice node $T_{[p]}$ is associated with the link $l$. If there is no other choice node in $p$ linked to $l$, we know by property 2 that there is a vector $v$ such that $v$ is admitted by $l$ and the value of $v$ on the choice node is $i$. Because of the independence of the other links with $l$, there is a tree in $T$ which

corresponds to the choice $i$ in $p$, and necessarily this tree is not in $S'$. If there is a choice node in $p$ linked to $l$, say at path $q$. There is a $j$ such that $qj \preceq p$. By induction on the number of choice nodes linked to $l$ along $p$, and by the same argument as above, we show that there is an element of the choice space that leads to $q$ and allows the choice $j$. But then, by property 1, such a choice allows the choice of $i$ at $p$. Once again, we have a tree in $T$ which is not in $S'$. □

## 4 Tree Schemata and Abstract Interpretation

Tree schemata were designed to be used in abstract interpretation. In this section, we show what is gained by this choice, and how abstract interpretation can deal with tree schemata.

### 4.1 Properties of Tree Schemata

**Expressiveness** One of the interesting properties of tree schemata is that they are more expressive than their most serious opponents, tree automata. Of course, tree schemata can easily express sets containing infinite trees, and even complex ones, but even when restricted to finite trees, the second example of Fig 3 shows that tree schemata can express some sets of trees which cannot be represented by tree automata.

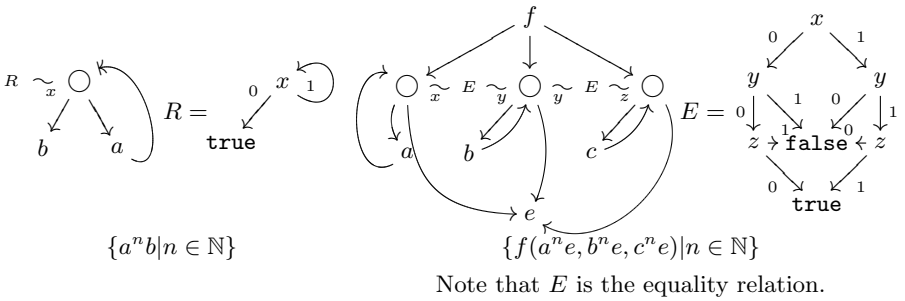
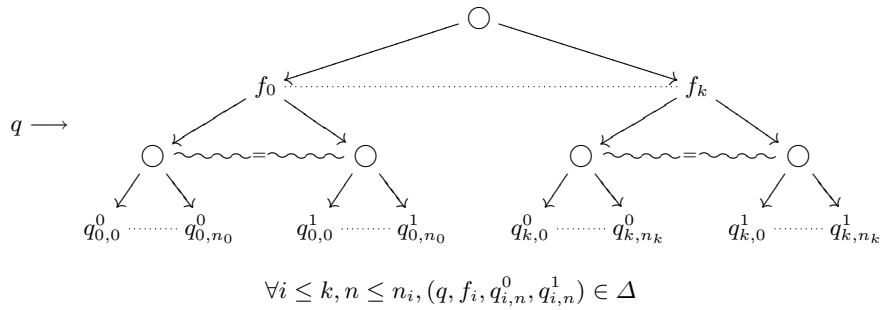



**Fig. 3.** Examples of Tree Schemata

With the appropriate representation for relations, we can also represent any regular set of trees with a tree schema. We give hereafter an idea of the construction. Let $L$ be the set of binary trees accepted by the finite top-down non-deterministic tree automaton, $\mathcal{A} = (Q, A, q_0, \Delta, F)$

(see [12] for a definition). To build the tree schema representing $L$, the first step is to build a non valid tree schema based on a non valid skeleton, but which represents $L$, and then to apply some rules that give a valid tree schema, without changing its meaning. The first graph is built using the rules of Fig 4 and connecting the states together. For the final states, we just add the labels of arity 0 to the first choice. For any tree $\overset{f}{\underset{t_0}{\swarrow}\ \underset{t_1}{\searrow}}$ recognized by the automaton starting at $q$, there is a rule $(q, f, q^0, q^1) \in \Delta$ such that each $t_i$ is recognized by the automaton starting at $q^i$. According to the pseudo decision procedure, it means that the tree is accepted by the tree schema starting at the choice node pointed by $q$, and the converse holds because of the relations $=$ which force a valid $(q^0, q^1)$ to be taken.




$$\forall i \leq k, n \leq n_i, (q, f_i, q^0_{i,n}, q^1_{i,n}) \in \Delta$$

**Fig. 4.** Rules to build the non valid tree schema

In order to simplify the skeleton on which the non valid tree schema is built, we can suppress choice nodes everywhere there is only one outgoing edge, but we still have some possible cascading choices, one of them with a relation, which cannot so easily be simplified. Fig 5 shows how this case can be reduced, by choosing the set of $S^0$'s and $S^1$'s to be exactly the sets of $T^0$'s and $T^1$'s, but without repetitions, and the relation $R$ to be $\{(a, b) | \exists c, d, e$ such that $S^0_a = T^0_{c,d}$ and $S^1_b = T^1_{c,e}\}$. The relation $R$ is finite, and so easy to represent with the techniques of [16]. The last step will combine the relations to make the skeleton deterministic: for each choice node such that there is an $S_i$ and an $S_j$ starting with the same label, we must merge the two schemata and incorporate their choice nodes in $R$. The immediate looping in the schema will result in the construction of infinite relations.

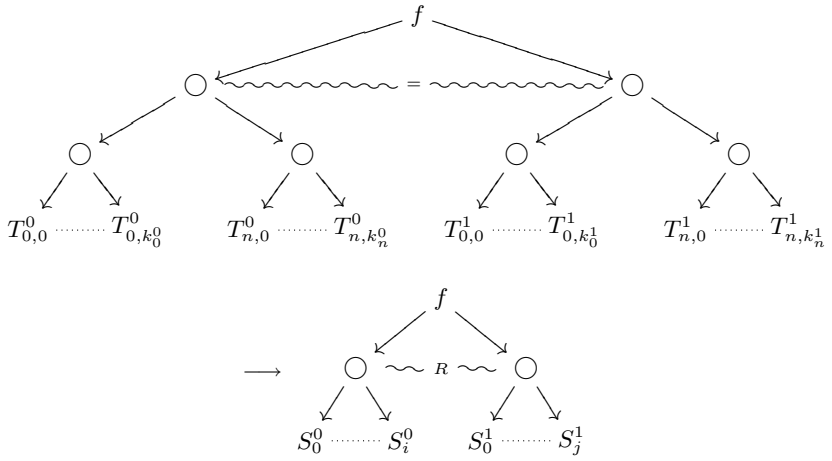



**Fig. 5.** Simplification rule to eliminate cascading choices

**Other Properties** Deciding the inclusion of tree schemata can be efficiently implemented. If the relations used in tree schemata are closed by union intersection and projection, then tree schemata are closed by union, intersection and projection. See [17] for proofs and algorithms. It seems that BDGs are the best suited so far to represent relations in tree schemata, and we will use them in the example of section 5. But as BDGs are not closed by union, tree schemata using BDGs are not closed by union, although we can indeed compute a best approximation (for set inclusion) of the union of two tree schemata.

Concerning the limits of tree schemata, it seems that we cannot represent the set of balanced trees, or the set $\left\{ \circlearrowright f \searrow t \;\middle|\; t \text{ is a tree} \right\}$ because it would require an infinite number of entry names in the relation denoting the equality between the infinite number of trees.

### 4.2 Interactions with Abstract Interpretation

Abstract interpretation deals with concrete and abstract domains to describe different semantics of programs. The semantics is generally computed via the resolution of a fixpoint equation. Such equations can be expressed with formal language transformers [9] using unions and projection (which subsumes intersection). The fixpoint can then be computed by an iteration sequence, possibly with widening. Such iteration can be

computed with tree schemata, where the approximation for union can be seen as a widening. One of the most common operations is the inclusion testing to decide whether we have reached a post-fixpoint. And inclusion testing is quite efficient with tree schemata.

The structure of tree schemata can easily be used to perform meaningful approximations (using widening techniques) when the size of the schemata is too big, as this size often comes from the relations, and we can choose to relax some relations. We can also simplify the skeletons if necessary.

One limitation of tree schemata is the finite number of labels for the trees. In the next section, we will see how an infinite domain can be approximated by a finite partition.

## 5 Example: Proving Fair Termination

In order to show the interests of one of the features of tree schemata —the ability to deal with infinite trees—, we chose a problem where using tree schemata can simplify a lot of things. We show how to prove automatically the termination under fairness assumption of concurrent processes with shared variables using abstract interpretation.

### 5.1 Semantics of the Shared Variables Language

We choose a simple language originated from [19] to describe concurrent processes sharing their variables. A program will be of the form $P := I; [P_1 || \ldots || P_n]; T$, where $I$, $P_i$ and $T$ are sequential deterministic programs composed of assignments of integers or booleans, `if-then-else` branching and `while` loops. In addition, the parallel processes $P_i$ have an await instruction of the form `await` $B$ `then` $S$ `end` where $B$ is a boolean expression and $S$ a sequential program without `await` instruction.

Informally the semantics of the program uses a global state. It executes $I$, and when $I$ ends each $P_i$ are executed in parallel with a notion of atomic actions which cannot interact (no simultaneous assignment to the same variable). The effect of the `await` instruction is to execute its program as an atomic action starting at a time when the boolean expression is true. The boolean expression is guaranteed to be true when the sequential program starts. Finally, when every parallel program has terminated, the program executes $T$.

We give the notion of atomic actions through a relation $\rightarrow$ defined by structural induction (following [11]). The definition is described in

$$\langle x := e, \sigma\rangle \to \langle E, \sigma[e/x]\rangle \qquad E;S = S;E = S$$

$$\frac{\sigma \models B}{\langle \texttt{while } B \texttt{ do } S, \sigma\rangle \to \langle S;\texttt{while } B \texttt{ do } S, \sigma\rangle} \qquad \frac{\sigma \models \neg B}{\langle \texttt{while } B \texttt{ do } S, \sigma\rangle \to \langle E, \sigma\rangle}$$

$$\frac{\sigma \models B \text{ and } \langle S, \sigma\rangle \to^* \langle E, \tau\rangle}{\langle \texttt{await } B \texttt{ then } S \texttt{ end}, \sigma\rangle \to \langle E, \tau\rangle} \qquad \frac{\langle S_1, \sigma\rangle \to \langle S_2, \tau\rangle}{\langle S_1;S, \sigma\rangle \to \langle S_2;S, \tau\rangle}$$

$$\frac{\langle P_i, \sigma\rangle \to \langle P_i', \tau\rangle}{\langle [P_1||\ldots||P_n], \sigma\rangle \to \langle [P_1||\ldots||P_{i-1}||P_i'||P_{i+1}||\ldots||P_n], \tau\rangle}$$

**Fig. 6.** Definition of the Transition Relation $\to$

figure 6 using a special empty program $E$. Based on this relation, we can define a semantics based on interleaving traces. We incorporate a notion of program points in the states. The program points of the parallel programs are the vectors of their program points. We have the following definition of the semantics $\mathcal{T}(\langle i : S, \sigma\rangle)$ of a program point $i$ with expression $S$ and environment $\sigma$:

$$\mathcal{T}(\langle i : S, \sigma\rangle) \stackrel{\text{def}}{=} \left\{ \begin{array}{c} \langle i,\sigma\rangle \\ \downarrow \\ t \end{array} \;\middle|\; t \in \mathcal{T}(\langle j : P, \tau\rangle) \text{ and } \langle S, \sigma\rangle \to \langle P, \tau\rangle \right\}$$

$$\mathcal{T}(\langle i : S, \sigma\rangle) \stackrel{\text{def}}{=} \langle i, \sigma\rangle \text{ if there is no state reachable from } \langle S, \sigma\rangle$$

A program $P$ is said to be *terminating* if and only if for every $\sigma$ $\mathcal{T}(\langle P, \sigma\rangle)$ does not contain any infinite trace. We can also define a deadlock as the end of a trace with index different from the last index of the program. We define $\mathcal{T}(P)$ as the union of the $\mathcal{T}(\langle P, \sigma\rangle)$ for all environment $\sigma$. The elements of $\mathcal{T}(P)$ are called the traces of $P$.

### 5.2 Expressing Program Properties as Sets of Traces

It is possible to express many program properties using just sets of traces. For example, termination is expressed as the set of all finite traces. To check that the program terminates, we just have to check that its set of traces is included in the termination property. In the same way, we can express termination without deadlock.

We can also express different kinds of fairness to decide whether a given trace of the program satisfies the fairness property. Every fairness property contains all finite traces. If it is an unconditional fairness [11]

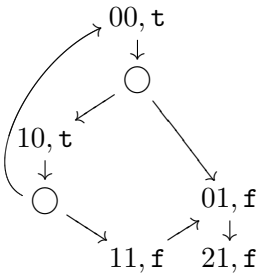
property then it contains also the infinite traces either with a finite passage in the concurrent part of the program, or such that each concurrent program that is not terminated progresses infinitely often.

We can prove that a program fairly terminates by proving that its set of traces intersected with the set of fair traces is included in the set of terminating traces.

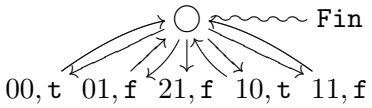
*Example 3.* Consider the program

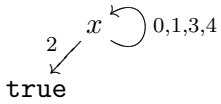
$$P =_0 b\texttt{:=true}[_0\texttt{while } b \texttt{ do } {}_1\texttt{skip}_2||_0 b\texttt{:=false}_1]_1$$

The set of traces of $P$ can be described by the following tree schema (we omit the beginning, which is not important):

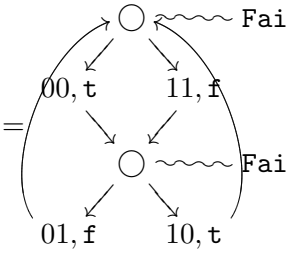
$\mathcal{T}(P) =$ 

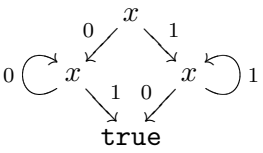

The termination property is expressed as:

Term = 

where Fin = 


To express the fairness property, we first describe the infinite fair traces, then we add the finite traces[3]:

$\text{Fair}_\omega =$ 

where Fin = 


$$\text{Fair} = \text{Fair}_\omega \cup \text{Term}$$

Then, to prove the fair termination of the program $P$, we just have to compute $\text{Fair} \cap \mathcal{T}(P)$ and verify that it is included in Term.

[3] We use this presentation just for the clarity of the schemata, we could just as well define Fair directly.

### 5.3 Abstraction of the set of traces

One of the limitations of tree schemata (necessary for a finite representation) is that we need a finite set of labels. Choosing the states to be the labels, we can have infinite sets of labels. To cope with this difficulty, we define an abstract semantics which approximates the concrete one described above, using the techniques of abstract interpretation.

Because we are interested in the control flow of the program, we just need to distinguish between states that evaluate differently on the boolean expressions in the program we analyze. We define abstract states to be each such partition of the set of states. We write $\texttt{states}^\sharp$ to denote this set of states. We define now abstract traces as traces labeled by $\texttt{states}^\sharp$. The concrete semantics is a set of concrete traces, the abstract semantics is a set of abstract traces. There is a Galois connection (for set of traces inclusion) [7] between those two semantics. Let `trace` be the set of sets of concrete traces, and $\texttt{trace}^\sharp$ be the set of sets of abstract traces. The concretisation of a trace $t^\sharp$ is the set of traces obtained by replacing every abstract state by a concrete state in the set of states it defines. The concretisation of a set of abstract traces is the union of the concretisations of its elements.

Sets of abstract traces are represented as tree schemata, but for our analysis to be ready, we need also to translate the properties into sets of abstract traces which will then be represented by tree schemata. The problem is that, whereas the fairness property can safely be over-approximated, we cannot over-approximate the termination property. The good news is that we can always represent this property *exactly*. Because of the way we chose the abstract states, the set of states with no successor for $\rightarrow$ is represented exactly by the set of abstract state with no successor. Thus the concretisation of the set of finite abstract traces is exactly the set of finite concrete traces. The set of finite abstract traces can easily be represented by a tree schema, the general method is the same as in the previous example.

For more powerful results, we need also to take into account the decreasing chains of integers in the states. For our purpose, such decreasing chains can be seen as a further constraint that some loop can only be taken finitely often, a fact that can be exactly expressed with tree schemata.

Of course, even with that analysis, we still manipulate abstractions of the sets of traces, so there will be some programs fairly terminating and not proved by this technique. This is inherent to approximation techniques, and unavoidable anyway when dealing with termination.

*Example 4.* Let $P$ be the following program:

$$P = {}_0x\texttt{:=}?_1; b\texttt{:=true}; [{}_0\texttt{while } b \texttt{ do } {}_1x\texttt{:=}x - 1_2 || \\ {}_0\texttt{await } x < 0 \texttt{ then } {}_1b\texttt{:=false end}_2]$$

In this example, the set of abstract states is $\{(x \geq 0, \texttt{t}), (x \geq 0, \texttt{f}), (x < 0, \texttt{t}), (x < 0, \texttt{f})\}$ to which we add the indexes of the program. The abstract state corresponding to the set of all the sets which are terminating is $(22, x < 0, \texttt{f})$.

Due to this approximation, the two possible states following $(10, x \geq 0, \texttt{t})$ are $(20, x \geq 0, \texttt{t})$ and $(20, x < 0, \texttt{t})$. The first state leads to a loop towards $(00, x \geq 0, \texttt{t})$. It is a very simple analysis that reveals that in this loop we have a decreasing chain, so this loop cannot be taken for ever. By adding this constraint we can perform the same analysis as in the previous example and still conclude that the program fairly terminates.

## 6 Conclusion

We presented a new representation for sets of trees. This representation has been developed with tractability in mind. It is based on a structure, the skeleton, which is an upper approximation of the set we represent. Tree schemata benefit from the great efficiency of the operations on skeletons. The skeletons are enriched with possibly infinite relations. With them, they are more powerful than tree automata, while more adapted to approximation techniques.

The example of fair termination showed that with such expressiveness it is possible to model very easily the behavior of programs. There was no need for complicated program transformations, introduction of variables or deep proofs. It is to be noted that the full power of tree schemata have not been used in this example, as no relation between distinct traces occurs.

The main drawbacks of this representation is that it is not fully tested yet. But the algorithms presented in [17] show that it is very promising, due to the unique representation of many elements of tree schemata. Moreover, the canonical decomposition of sets of trees in a tree structure and relations, allows for a very natural introduction of counters which can be very useful in analysis, especially if some of these counters are related to the programs we analyze.

**Aknowledgements**

I would like to thank the anonymous referees for their encouraging comments and their constructive remarks which helped a lot in the improvement of this paper.

## References

1. Aiken, A., and Murphy, B. R. Implementing regular tree expressions. In *Functional Programming Languages and Computer Architecture* (1991), J. Hughes, Ed., vol. 523 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 427–446.
2. Andersen, N. Approximating term rewriting systems with tree grammars. Tech. Rep. 86/16, Institute of Datalogy, University of Copenhagen, 1986.
3. Biehl, M., Klarlund, N., and Rauhe, T. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata* (1997), vol. 1260 of *Lecture Notes in Computer Science*.
4. Börstler, J., Möncke, U., and Wilhelm, R. Table compression for tree automata. *ACM Transactions on Programming Languages and Systems 13*, 3 (July 1991), 295–314.
5. Bryant, R. E. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35* (August 1986), 677–691.
6. Charatonik, W., and Podelski, A. Co-definite set constraints. In *9th International Conference on Rewriting Techniques and Applications* (March-April 1998), T. Nipkow, Ed., vol. 1379 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 211–225.
7. Cousot, P. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. PhD thesis, Université de Grenoble, March 1978.
8. Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL '77)* (1977), pp. 238–252.
9. Cousot, P., and Cousot, R. Formal languages, grammar and set-constraint-based program analysis by abstract interpretation. In *Conference on Functional Programming and Computer Architecture (FPCA '95)* (June 1995), pp. 170–181.
10. Devienne, P., Talbot, J., and Tison, S. Solving classes of set constraints with tree automata. In *3th International Conference on Principles and Practice of Constraint Programming* (October 1997), G. Smolka, Ed., vol. 1330 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 62–76.
11. Francez, N. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.
12. Gécseg, F., and Steinby, M. *Tree Automata*. Akadémia Kiadó, 1984.
13. Heintze, N. *Set Based Program Analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1992.
14. Henriksen, J. G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., and Sandholm, A. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems* (1996), vol. 1019 of *Lecture Notes in Computer Science*.

15. Jones, N. D., and Muchnick, S. S. Flow analysis and optimization of LISP-like structures. In *6th ACM Symposium on Principles of Programming Languages (POPL '79)* (January 1979), ACM Press, pp. 244–256.
16. Mauborgne, L. Binary decision graphs. In *Static Analyis Symposium (SAS'99)* (1999), A. Cortesi and G. Filé, Eds., vol. 1694 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 101–116.
17. Mauborgne, L. *Representation of Sets of Trees for Abstract Interpretation*. PhD thesis, École Polytechnique, Palaiseau, France, November 1999.
18. Mauborgne, L. Improving the representation of infinite trees to deal with sets of trees. In *European Symposium on Programming (ESOP 2000)* (2000), vol. to appear of *Lecture Notes in Computer Science*, Springer-Verlag.
19. Owicki, S., and Gries, D. Verifying properties of parallel programs: an axiomatic approach. *CACM 19*, 5 (August 1976), 279–286.
20. Reynolds, J. Automatic computation of data set definitions. In *Information Processing '68* (1969), Elsevier Science Publisher, pp. 456–461.
21. Schwartzbach, M. I. Infinite values in hierarchical imperative types. In *15th Colloquium on Trees in Algebra and Programming (CAAP '90)* (May 1990), A. Arnold, Ed., vol. 431 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 254–268.
22. Sørensen, M. H. A grammar-based data-flow analysis to stop deforestation. In *Trees in Algebra and Programming — CAAP '94* (April 1994), S. Tison, Ed., vol. 787 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 335–351.
23. Thatcher, J. W., and Wright, J. B. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory 2* (1968), 57–82.
24. Vardi, M. Y. Nontraditional applications of automata theory. In *Theoretical Aspects of Computer Software* (April 1994), M. Hagiya and J. C. Mitchell, Eds., vol. 789 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 575–597.