Nordic Journal of Computing

An Incremental Unique Representation for Regular Trees

Laurent Mauborgne
École Normale Supérieure

Abstract. In order to deal with infinite regular trees (or other pointed graph structures) efficiently, we give new algorithms to store such structures. The trees are stored in such a way that their representation is unique and shares substructures as much as possible. This maximal sharing allows substantial memory gain and speed up over previous techniques. For example, equality testing becomes constant time (instead of $O(n\log(n))$). The algorithms are incremental, and as such allow good reactive behavior. These new algorithms are then applied in a representation of sets of trees. The expressive power of this new representation is exactly what is needed by the original set-based analyses of Heintze and Jaffar [1990], or Heintze [1994].

CR Classification: See *Computing Revues*

Key words: Infinite Trees, Sharing, Tree Skeletons, Cartesian Approximation.

1. Introduction

1.1 Motivations

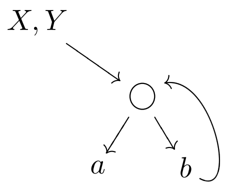
When applying set-based analysis techniques for practical applications, one is surprised to see that the representation of the sets of trees is not very efficient. For example, to represent the set b^*a, we can get the following representation (from the original paper of Heintze and Jaffar [1990]):

$$\begin{tikzcd}
& X \arrow[d] & & \\
& \bigcup \arrow[dl] \arrow[dr] & & \\
a & & b \arrow[d] & \\
& & Y \arrow[d] & \\
& & \bigcup \arrow[dl] \arrow[dr] & \\
& a & & b \arrow[uul, bend right]
\end{tikzcd}$$

And this is just one example of possible inefficiencies in the representation. Even when we use tree automata, we cannot overcome this problem without

Received August 27, 2004.

performing a minimization of the whole automaton at each step. We propose a new way of dealing with this kind of structure to get a representation that is as small as possible during the computation. The example above would give the following representation:



After analysis of the problem, it appears that the underlying structures we want to optimize can be described mathematically as regular infinite trees. Because tree structures appear everywhere in computer science where a hierarchy occurs, we found it interesting to present the algorithms in an independent way. In this way, our technique appears as an extension of an efficient solution to store finite trees. Another interest of infinite regular trees is that they are isomorphic to pointed graphs, that is graphs with a distinguished node (such as word automata e.g.).

1.2 The Choice of the Representation

The efficiency of a representation lies in its compactness and in the complexity of the algorithms using the objects. A first consequence is that we have to find a balance between those two aspects. A second consequence is that the choice of the representation depends on the intended use for the representation, so we cannot entirely abstract from our motivations. If we are to interpret the regular trees as sets, the algorithms we will have to perform can benefit from the reuse of intermediate results (the memoizing of Michie [1968]). So, to keep thing general, the operation we will try to optimize will be equality testing.

1.3 Incremental Uniqueness

Constant time equality testing can be achieve if the representation is *unique*. A representation is said to be unique if during a computation there can be no duplicate memory location for a given data. To test the equality of two objects, we just have to compare the memory location of their representation. For example, if the data are sets of words and their representations are finite automata, it is not enough to minimize the automata, because we can still have two equivalent automata at two different memory locations. We would have to perform minimization over the whole set of automata at each step, which would be quite costly. So we come to the second idea, incrementality, which is more difficult to quantify: we will try to do as few work as possible each time we modify the data. In this way, we hope to keep a good balance between the compactness of the representation and the complexity of the algorithms used to ensure the compactness.

$$\begin{tikzcd}
 & & f \arrow[dl, "0"'] \arrow[dr, "1"] & \\
 & f \arrow[dl, "0"'] \arrow[dr, "1"] & & g \arrow[d, "0"] \\
a & & a & a
\end{tikzcd}$$

is

$$\begin{cases} \varepsilon \to f, & 01 \to a, \\ 0 \to f, & 1 \to g, \\ 00 \to a, & 10 \to a \end{cases}$$

Fig. 1: The representation of a tree, and its mathematical definition

There exists an incremental unique representation for finite trees, which uses just the minimum amount of memory by sharing equivalent subtrees. This saves a lot of space. It is used, for example, with sets of words represented as a tree to share common prefixes. It has been the source of the success of the Binary Decision Diagrams (BDDs) of Bryant [1986], which are considered one of the best representations for boolean functions so far.

So we tried to extend these techniques to infinite trees. The problem was that, as soon as a loop occurs somewhere in the data, finite tree techniques are no longer adequate. The main contribution of this article is to extend the good results of unique sharing representation from finite trees to infinite trees. These techniques can be applied to the representation of sets of trees in set-based analysis (see section 6), but they can also be applied directly to the representation and manipulation of finite automata, or infinite boolean functions, as in Mauborgne [1999].

1.4 Roadmap

After a recollection of the classical results over finite trees in section 2, we present the solutions for the most difficult problems with infinite trees in the section 3 on cycles. The general problem is then treated in section 4, with a full example. Complexity issues and algorithms to manipulate infinite trees are discussed in section 5. The application to sets of trees implies the description of a new encoding to keep the uniqueness of the representation. This new contribution is described in section 6.

2. Classical Representation of Trees

2.1 Trees and Graphs

As we deal with the computer representation of data structures, we must give a clear meaning to the word “representation”, and in particular clearly distinguish between what is represented and what is the representation. For this reason, we will give a mathematical definition of what is a tree, and another one for the way it is usually stored in a computer (see Fig. 1 for an example of tree and its representation).

Let $\mathbb{N}^*$ be the set of words over $\mathbb{N}$, ε denoting the empty word. We note $\prec$ the prefix ordering on words and $u.v$ the concatenation of the words u and v. The alphabetic ordering on words over $\mathbb{N}$ will be denoted $\prec_\alpha$. Let F be a finite set of labels.

Definition 1. *A tree t labeled by F is a function of $E \to F$ such that $E \subset \mathbb{N}^*$ and $\forall\, p \in \mathbb{N}^*, \forall\, i \in \mathbb{N},\, p.i \in E \Rightarrow (p \in E$ and $\forall\, j < i,\, p.j \in E)$. We note $pos(t) \overset{def}{=} E$.*

Let $p \in pos(t)$. The subtree of t in p, written $t_{[p]}$ is defined by: $pos(t_{[p]}) \overset{\text{def}}{=} \{q \in \mathbb{N}^* \mid p.q \in pos(t)\}$, and $t_{[p]}(q) \overset{\text{def}}{=} t(p.q)$. A tree is uniquely determined by the label of its root, $t(\varepsilon)$, and by the children of the root, the different $t_{[i]}, i \in \mathbb{N}$. In the sequel, a generic tree will be denoted $\begin{matrix} & f & \\ \swarrow & & \searrow \\ t_0 & \cdots & t_{n-1} \end{matrix}$, where f is the label of the root, and $(t_i)_{i<n}$ are the children of the root.

When representing a tree in a computer, we usually use one computer location for each position p in $pos(t)$, where we store the label $t(p)$ and the location of the different children (the $p.i$'s in $pos(p)$) of this position. Such a representation is well modeled by a graph, where each node of the graph corresponds to a computer location. We do not give the most general definition of graphs, but the definition that is useful in this article to represent trees.

Definition 2. *A* graph *G labeled by F is composed of two finite sets, the node set, G^N, and the edge set, $G^E \subset G^N \times G^N \times \mathbb{N}$, and every node of the graph is associated with a label in F.*

A *path* in the graph G is a sequence $N_0 \dots N_k$ of nodes in G^N, such that $\forall l < k$, $\exists i\ (N_l, N_{l+1}, i) \in G^E$. The node N_k is called the end of the path.

We define the notion of path label in a graph: let $p \in \mathbb{N}^*$, p is a *path label* from the node N if and only if $p = \varepsilon$ or $p = i.q$ and there is an $M \in G^N$ such that $(N, M, i) \in G^E$ and q is a path label from M. Note that at least one path corresponds to such a path label. If there is only one path corresponding to the path label p from the node N, and O is the node at the end of the path, we write $N.p = O$ (otherwise, $N.p$ is not defined).

We define $\mathcal{G}(N)$ as the graph defined by the nodes which can be reached from N. We will often identify a node N and the graph $\mathcal{G}(N)$.

Definition 3. *A node N* represents *a tree t if and only if the set of path labels from N is exactly $pos(t)$, and $\forall p \in pos(t)$, $N.p$ is well defined, and its label is $t(p)$.*

Because we are interested in tree representation, in the sequel, we will assume that every node of our graphs represent a tree.

$$t = \begin{cases} (10)^* & \rightarrow \ f \\ (10)^*0 & \rightarrow \ a \\ 1(01)^* & \rightarrow \ g \end{cases} \qquad \text{can be represented by}$$

$$\begin{tikzcd} & f \arrow[dl, "0"'] \arrow[dr, "1"] & \\ a & & g \arrow[ul, bend right=60, "0"'] \end{tikzcd}$$

Fig. 2: An infinite regular tree

A *finite tree* t is a tree such that $pos(t)$ is finite. There is always a possible representation by a finite graph for finite trees. In the most common use, one node corresponds to each path of the finite tree.

A *regular* tree t is a tree such that the number of distinct subtrees of t is finite. Such a tree can be infinite, but it can still be represented by a finite graph [Colmerauer 1982], see Fig. 2 for an example.

2.2 Best Graph Representation

The naive representation, which consists in using any graph representing the tree [Colmerauer 1982], is very easy to deal with and quite widely used for small problems. But we can do far better if we observe that some nodes can represent different paths of the tree, as long as the subtrees at these paths are the same. This is called sharing the subtrees (see e.g. Aho *et al.* [1983]). In fact, the best we can do is to have exactly one node for each distinct subtree. This is what we call the best graph representation of a tree (which may not be the best possible representation in the sense discussed in section 1.2). In the case of finite trees, this can save a lot of space, and even time by memoizing [Michie 1968], and in the case of infinite regular trees, we avoid the possibility of unbounded representation for a given tree.

When dealing with many trees, we can do even better: considering the entire computer memory as one graph, we can optimize the representation for all the trees, and have in effect exactly one memory location for each distinct tree we need to store. An immediate consequence is that we just have to compare the location of the roots (the node representing the trees) to compare entire trees. Such a technique is used e.g. in BDDs [Bryant 1986] to achieve impressive speed up and memory gain.

The technique to obtain the best graph representation of the trees uses a dictionary mechanism linking keys to nodes of the graph, usually a hash table. The keys are built incrementally: if the keys for the $(t_i)_{i<n}$ are known and linked to the nodes $(N_i)_{i<n}$, then the key for $\begin{smallmatrix} & f & \\ \swarrow & & \searrow \\ t_0 & \cdots & t_{n-1} \end{smallmatrix}$ is $(f, (N_i)_{i<n})$. Each time a key is not present in the dictionary, it is associated with a new node N, with edges to the N_i's. If we come to a tree whose key is already in the dictionary, we use the corresponding node. As the trees are always built from leaves to root, we have indeed a best graph representation for the trees.

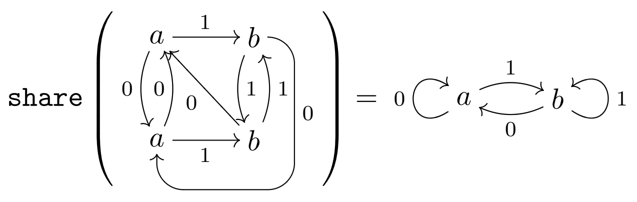


Fig. 3: Application of the `share` algorithm.

3. Dealing with Cycles

When representing infinite trees though, we cannot go from the leaves to the root, so we cannot start the key mechanism which leads to the best graph representation. The difficulty lies in the infinite paths of the tree, that is the cycles of the graph representing the tree. Whereas in finite trees there is no need to see beyond the immediate children of a given node, when dealing with cycles, we can have reasons to look further, in order to detect the two causes of cycle unfolding: cycle growth and root unfolding. For example, consider the cycle $\begin{tikzcd} a \arrow[r, bend right] & b \arrow[l, bend right] \end{tikzcd}$. Then $\begin{tikzcd} & b \arrow[dl, bend right] & \\ a \arrow[dr, bend right] & & a \arrow[ul, bend right] \\ & b \arrow[ur, bend right] & \end{tikzcd}$ is an example of cycle growth, and $\begin{tikzcd} a \arrow[r] & b \arrow[r, bend right] & a \arrow[l, bend right] \end{tikzcd}$ is an example of root unfolding. In this very simple example, it is easy to reduce root unfolding by looking at the key of the root, but it is much more difficult if the root itself is still in another cycle. In order to concentrate on the real difficulties, we suppose in this section that we deal with strongly connected graphs, that is graphs such that there is a path between any pair of nodes.

3.1 Cycle Growth and Tree Keys

We give $\equiv_{\text{tree}}$ as the equivalence between nodes representing the same tree. The goal of cycle growth reduction is to find an equivalent graph with the minimum number of nodes.

Definition 4. *A graph G is said to have a minimal number of nodes if, whatever the nodes N and M of G, $N \equiv_{\text{tree}} M \Rightarrow N = M$.*

The problem of finding the equivalent graph with minimum number of nodes is a partitioning problem. The Partitioning Problem has been solved in time $n\log(n)$ by Hopcroft [1971] for finite automata (which are special cases of graphs), and in the general case by Cardon and Crochemore [1982]. We call `share`(N) the algorithm that takes a node N and modifies the associated graph so that it has the fewest possible nodes (see Fig. 3 for an example, and Appendix A.1 for a description).

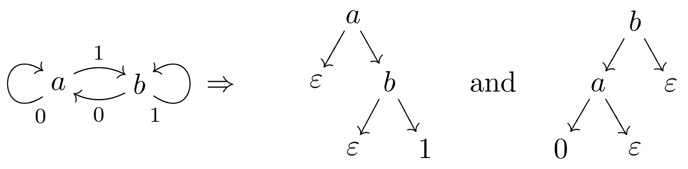


Fig. 4: A graph, followed by the tree keys of its two nodes

Cycle growth reduction corresponds to the state of the art in automata representation. But we want to go further: we need that the representation be unique whatever the different versions of the same tree. To perform this, we give a key which distinguishes between non isomorphic graphs. This key is associated with a given node N of the graph. It is a finite tree which corresponds to the graph as long as we do not loop, but as soon as we loop, the label of the node is replaced by its path label from N. It is described as $\texttt{treeKey}(N)$. See Fig. 4 for an example, and Appendix A.2 for a description. In order to give a formal definition of this tree, we first define the set of non looping path labels from N: $\text{NoCycle}(N) \stackrel{\text{def}}{=} \{p \mid N.p \text{ is defined, and } \nexists q \prec p,\ N.q = N.p\}$. The tree $\texttt{treeKey}(N)$ is defined by:

$$\begin{aligned}
pos(\texttt{treeKey}(N)) &\stackrel{\text{def}}{=} \{p.i \mid p \in \text{NoCycle}(N) \wedge (N.p, M, i) \in \mathcal{G}(N)\} \cup \{\varepsilon\} \\
\texttt{treeKey}(N)(p) &\stackrel{\text{def}}{=} \text{the label of } N.p \text{ if } p \in \text{NoCycle}(N) \\
&\quad\ \inf_{\prec_\alpha} \{q \mid N.q = N.p\} \text{ otherwise}
\end{aligned}$$

The isomorphism between graphs is not the same thing as $\equiv_{\text{tree}}$. In general it can differentiate two graphs which represent the same tree. The interesting point is that it is indeed the same relation on graphs with a minimal number of nodes.

Proposition 1. *Whatever M and N, such that $\mathcal{G}(M)$ and $\mathcal{G}(N)$ are graphs with minimal number of nodes, $\textit{treeKey}(M) = \textit{treeKey}(N) \Leftrightarrow M \equiv_{\text{tree}} N$.*

Proof. The difficult point is $M \equiv_{\text{tree}} N \Rightarrow \texttt{treeKey}(M) = \texttt{treeKey}(N)$. Suppose there are M and N such that $\mathcal{G}(M)$ and $\mathcal{G}(N)$ are graphs with minimal number of nodes, $M \equiv_{\text{tree}} N$ and $\texttt{treeKey}(M) \neq \texttt{treeKey}(N)$. Let $t_M = \texttt{treeKey}(M)$ and $t_N = \texttt{treeKey}(N)$. Because $t_M \neq t_N$, there is a path p such that $t_M(p) \neq t_N(p)$. But if $t_M(p)$ is a label of the graph, $t_M(p)$ is the label of $M.p$, and the same holds for N. Because $M \equiv_{\text{tree}} N$, $M.p$ and $N.p$ have the same label, so at least one of $t_M(p)$ or $t_N(p)$ is not a label of the graphs (and so is in $\mathbb{N}^*$), say $t_M(p)$. It means there is a $q \prec p$ such that $M.q \equiv_{\text{tree}} M.p$. So $N.q \equiv_{\text{tree}} N.p$, but by minimality of the number of nodes of $\mathcal{G}(N)$, $N.q$ and $N.p$ must be the same node, and so $t_N(p) = q = t_M(p)$. $\square$

Because we can find an equivalent graph with minimal number of nodes for strongly connected graphs, we have a valid key mechanism for any strongly connected graph: we first apply `share`, then `treeKey`.

3.2 Root Unfolding and Partial Keys

With just `share` and `treeKey` (applied to every node), we can have a unique representation that shares common subtrees. But as we need to start the whole process from the beginning for each little modification in the trees, such a process would be quite slow. Moreover, it is much better to apply the `share` algorithm on the smallest possible graphs. As it is not a linear algorithm, we have better results if we can split the graph and apply the algorithm to each separate subgraph only.

The finite parts of the tree can always be treated in the classical way, while the loops will need a special treatment. In order to decompose the graph and mark those parts of the graph which have been definitely treated, we introduce partial keys. A partial key looks like a node key for a finite tree, a label followed by a vector of nodes, except that for some parts of the vector, there is no node (see section 4.3 for an example). A partial key k has a name: $\mathrm{name}(k) \in F$ and is a partial function from $\mathbb{N}$ to nodes. A graph labeled by partial keys is such that for every node N in the graph, if k is the partial key for N, the edges in the graph correspond to those integers for which the partial key is not defined. For example, if a node is labeled by f of arity 3, we can have a partial key which is not defined on 0 and 1 (we write a $\bullet$), and on 2 its value is the node M_4. We write $(f, \bullet\bullet M_4)$ for this partial key. The only edges that can leave from such a node would be labeled by 0 and 1. The idea is that what is in the partial keys is uniquely represented. In our example, the node M_4 is a unique representation of some tree. Later on during the computation, it is possible that we have a unique representation for the first component, say with node M_2, and the partial key becomes $(f, M_2 \bullet M_4)$. When a partial key is full (defined everywhere), then the node should be a unique representation.

This new graphs have new equivalence relation, $\equiv_{\mathrm{pk}}$ which is implied by $\equiv_{\mathrm{tree}}$. This new equivalence relation corresponds to $\equiv_{\mathrm{tree}}$ after the expansion of the partial keys in the graph. In order to give an exact meaning to the expansion of partial keys, we start with some notations. If the partial key label of a node M is $(f, (pk_i)_{i<k_M})$, we write $\mathrm{pkla}(M) = k_M$ for the partial key label arity of M, and $\forall i < \mathrm{pkla}(M)$, $\mathrm{pklb}(M, i) = pk_i$ is the value of the partial key on i (which we write $\bullet$ if it is undefined). We write also $\mathrm{pkld}(M) \overset{\mathrm{def}}{=} \{i < \mathrm{pkla}(M) \mid \mathrm{pklb}(M, i) \neq \bullet\}$ the set of indexes on which the partial key is defined. Given a node N, defining a graph labeled by partial keys $\mathcal{G}(N)$, we define `expand`(N) as the corresponding node, defining a graph

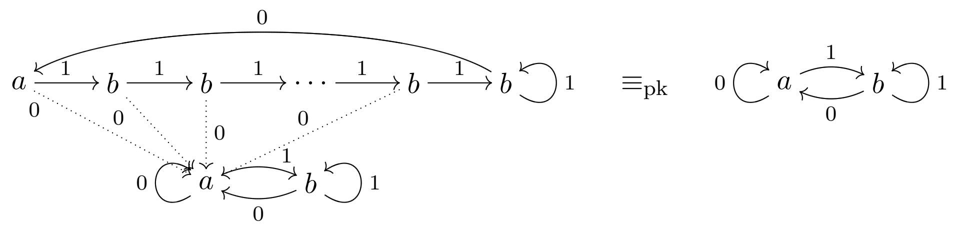


Fig. 5: Root unfolding of a cycle

labeled by F, $\mathcal{G}(\texttt{expand}(N))$. This graph is defined as:

$$\begin{aligned}
\mathcal{G}(\texttt{expand}(N))^N &\overset{\text{def}}{=} \mathcal{G}(N)^N \cup \bigcup_{M\in\mathcal{G}(N)^N} \bigcup_{i\in\text{pkld}(M)} \mathcal{G}(\text{pklb}(M,i))^N \\
\mathcal{G}(\texttt{expand}(N))^E &\overset{\text{def}}{=} \mathcal{G}(N)^E \cup \bigcup_{M\in\mathcal{G}(N)^N} \\
&\qquad \left(\bigcup_{i\in\text{pkld}(M)} \{(M, \text{pklb}(M,i), i)\} \cup \mathcal{G}(\text{pklb}(M,i))^E \right)
\end{aligned}$$

The new labels in the expanded graphs are just the names of the partial keys. The exact definition of $\equiv_{\text{pk}}$ is: $M \equiv_{\text{pk}} N$ if and only if $\texttt{expand}(M) \equiv_{\text{tree}} \texttt{expand}(N)$.

But now, with those partial keys, we can have a strongly connected graph such that, by root unfolding, one of its nodes is equivalent to a node in a partial key. Fig. 5 shows a case of root unfolding, which can be as big as we want, even after cycle growth reduction. In this figure, dotted lines correspond to nodes stored in partial keys. So, we must look for such a node, even before applying the `share` algorithm.

The name of the algorithm performing this task is $\texttt{shareWithDone}(N)$. It returns N if and only if no other node in the partial keys is equivalent to N. Otherwise, it returns the node in the partial keys that is equivalent to N. This algorithm uses some properties of the graph to reduce the complexity of the computation. Let G be the graph associated with N. As always in this section, we suppose that G is strongly connected. We call H the graph already computed and that is reachable from the partial keys of G. The algorithm determines whether a node of G is equivalent to a node of H. If it is the case, then there is root unfolding. If not, there is no root unfolding. We show that it is enough to verify this property for one node to treat the entire graph G because G is strongly connected. Suppose N is equivalent to M in H. Then, whatever the path label from N, p, $N.p$ is equivalent to $M.p$. Because H has been treated already, any $M.p$ is in H, and because G is strongly connected, any node of G is an $N.p$.

There is a kind of reciprocal property that is exploited too: for some subsets of H^N, if no node of the subset is equivalent to a particular node of G, then they are not equivalent to any node of G. A subset of H^N is said to be closed if and only if, for every path label p from every node N in the subset, $N.p$ is in the subset.

Proposition 2. *$\forall H' \subset H^N$ such that H' is closed, if $\exists N \in G^N$ such that $\forall M \in H'$, $N \not\equiv_{\text{pk}} M$, then this holds for every $N \in G^N$.*

Proof. Let H' be such a subset and N a node of G. If N is not equivalent to any node in H', then, suppose there is an $M \in G^N$ and an $O \in H'$ such that M is equivalent to O. As G is strongly connected, there is a p such that$M.p = N$. So, N would be equivalent to $O.p$, which is in H'. This proves that no element of G^N is equivalent to any element of H'. $\square$

Because of these properties, we can use the following algorithm (see Appendix A.3) for `shareWithDone`: we just compare every nodes of G with the nodes that are reachable from their partial keys and not already encountered. This comparison can be quite efficient by exploiting the fact that the nodes in the partial keys are unique representations of trees, although we have a quadratic worst case complexity.

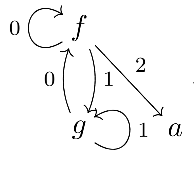
We will show in the next section, that by applying first `shareWithDone`, then `share` and then `treeKey`, we can indeed represent uniquely (and with the least possible number of nodes) any strongly connected graph, in an incremental process.

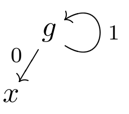
4. The Best Graph Representation for Infinite Trees

As we defined it page 5, the best graph representation for an infinite tree is a graph with minimal number of nodes.

4.1 Informal Presentation

In order to show how we can produce the best graph representation for an infinite tree, we solve the following generic problem: considering a graph representing a tree t, return an equivalent graph with a minimal number of nodes. Such a problem is generic, because whatever the way we build a tree, we can simulate the history of the construction of the tree while keeping the graph minimal by the incremental minimization of a bigger graph. That is the reason why we can derive many algorithms on trees from this problem. Suppose for example that we want to compute the tree t computed in the following way:

$t_2 =$ 

, $t_5 =$ 

, $t_1 = f(t_2, t_5, a)$

and t is t_1 where x is replaced by a loop to the root (t). Then we can simulate this way of computing t by the graph of Fig. 6, which is bigger than the best graph representation. The way this graph is transformed in an incremental way is described section 4.3.

In order to produce the minimal graph in an incremental way, we use two dictionary mechanisms and a decomposition of the graph. First, we apply the classical algorithm, using the dictionary D, on the finite subtrees of the tree. When a finite subtree is entirely treated, it is incorporated in the graph through partial keys. Second, when there is no more finite subtree, there is a subtree represented by a strongly connected graph. The dictionary D_G stores the tree keys of such graphs, and after `shareWithDone` and if necessary, `share`, we can decide whether another equivalent graph has already been encountered, and if not, use new nodes. When the strongly connected graph is treated, it is considered as just a node, and so we can iterate on our algorithm until we give the representation of the root.

4.2 The Algorithm

The algorithm maps a node representing the tree t into the node of a new graph representing t with minimal number of nodes, and in an incremental way.

We suppose given a dictionary D which maps full keys to nodes corresponding to a unique representation of the associated tree, and a dictionary D_G which maps tree keys (in fact keys of these finite trees) to nodes corresponding to a unique representation of the associated strongly connected graph.

The algorithm uses local dictionaries too, which we assume to be empty when the process starts on a tree. The dictionary `encountered` contains the nodes of the original representation already encountered, associated with their computed nodes (so that we do not loop). The set `returnNodes` is used to detect the roots of the loops.

A node is considered "treated" when it is in the dictionary D (and so it represents uniquely a tree). To decide whether a node is "treated", we just have to look at its key: it is "treated" if the key is full.

```
representation($N$)
  Step 1  **if** $N \in dom$(encountered) **then**
              **if** encountered($N$) is not treated add it in returnNodes
              **return** encountered($N$)
  Step 2  $M$ is a new node labeled by the empty partial key $k$ of name
                    the label of $N$. Add $N \rightarrow M$ to encountered.
  Step 3  **for** each child $N_i$ of $N$ **do**
          3a  $M_i \leftarrow$ representation($N_i$)
          3b  **if** $M_i$ is treated, **then** add it to $k$
              **else** $M.i \leftarrow M_i$
  Step 4  **if** $k$ is full **then**
```

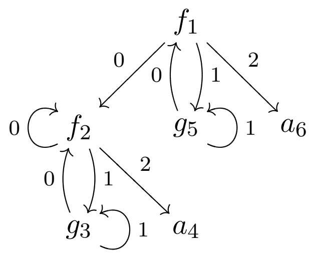


Fig. 6: Example

```
                if $k \in D$ **then** encountered($N$) $\leftarrow$ $D(k)$ and **return** $D(k)$
                **else** add $k \rightarrow M$ to $D$ and **return** $M$
  Step 5  remove $M$ from returnNodes
  Step 6  **if** returnNodes $= \emptyset$ **then return** representCycle($M$)
  Step 7  **return** $M$

representCycle($N$)
  Step 1  **if** shareWithDone($N$) $\neq N$ **then return** shareWithDone($N$)
  Step 2  share($N$)
  Step 3  **if** treeKey($N$) $\in D_G$ **then**
                update encountered and **return** $D_G$(treeKey($N$))
  Step 4  **for** each node $M$ in the graph defined by $N$ **do**
          4a    add treeKey($M$) $\rightarrow M$ to $D_G$
          4b    add the children of $M$ to its partial key $m$
          4c    add $m \rightarrow M$ to $D$
  Step 5  **return** $N$
```

4.3 Example

We present the algorithm to represent regular trees on an example, the graph of Fig. 6, where each node is assigned a number. We will write N_i for the node number i of the initial representation, and M_i for the node created at step 2 of `representation`(N_i).

`representation`(N_1) calls `representation` for N_2, N_5 and N_6. The call to `representation` on N_2 will return the node M_2. It will also store various nodes in D, and in particular $(a) \rightarrow M_4$. The call on N_5 will just return an untreated node M_5, with nothing added in the dictionaries. The call on N_6 will recognize on step 4 that a is in D and so it will return M_4.

Thus, at step 5, `returnNodes` $= \{M_1\}$ becomes empty, and we call `re-`

`presentCycle` with the graph[1] $$\begin{tikzcd}(f, M_2 \bullet M_4) \arrow[d, bend left, "1"] \\ (g, \bullet\bullet) \arrow[u, bend left, "0"] \arrow[loop right, "1"]\end{tikzcd}$$. A call to `shareWithDone` returns the node M_2. So the return value of `representation` on N_1 is M_2, the node labeled by f in the graph $$\begin{tikzcd} f_2 \arrow[loop left, "0"] \arrow[d, bend left, "1"] \arrow[dr, "2"] & \\ g_3 \arrow[u, bend left, "0"] \arrow[loop right, "1"] & a_4 \end{tikzcd}$$. Moreover, the dictionaries will be:

$$D = \{(a) \to M_4, (g, M_3 M_2) \to M_3, (f, M_2 M_3 M_4) \to M_2\}$$

$$D_G = \left\{ \begin{tikzcd}[column sep=small] & (f, \bullet\bullet M_4) \arrow[dl] \arrow[d] & \\ \varepsilon & (g, \bullet\bullet) \arrow[dl] \arrow[dr] & \\ \varepsilon & & 1 \end{tikzcd} \to M_2, \quad \begin{tikzcd}[column sep=small] & (g, \bullet\bullet) \arrow[d] \arrow[dr] & \\ & (f, \bullet\bullet M_4) \arrow[dl] \arrow[dr] & \varepsilon \\ 1 & & \varepsilon \end{tikzcd} \to M_3 \right\}$$

4.4 Proof of the Algorithm

The algorithm returns the node of a graph. We must prove that this graph represents the same tree as the original graph, and that it is a graph of maximal sharing.

First, notice that the algorithm terminates, because of the dictionary `encountered` which implies that each node of the original graph is treated only once.

The correctness of the algorithm is derived from the fact that we return the same graph as the original, except when we recognize that an equivalent node had already been encountered (through the node keys or the tree keys), in which case we replace one node by the other. It is the case step 4 of `representation`, and steps 1, 2 and 3 of `representCycle`

The fact that the resulting graph has the minimal number of nodes lies in the use of the dictionaries D and D_G to ensure that we never duplicate any node. The dictionary D contains the node keys of every node encountered, and the dictionary D_G contains the tree key of every node of every strongly connected graph with minimal number of nodes we encounter. We can prove that each time we definitely introduce new nodes, there is no duplication. Definitive introduction is performed in two points: step 4 of `representation`, and step 4 of `representCycle`.

Step 4 of `representation`, we know that the key k is not in D. Moreover, each one of the M_i composing the key is unique because nodes in partial keys

[1] Remember that $(f, M_2 \bullet M_4)$ is the partial key which is not defined on its second component.

have already been treated. So if a tree $\begin{array}{c} f \\ \swarrow\ \searrow \\ t_0 \cdots t_{n-1} \end{array}$ had already been encountered, the key $(f, (M_i)_{i<n})$ would already have been encountered.

Step 4 of `representCycle`, we know that the key `treeKey`(`share`(N)) has never been encountered before. Because such a key is valid for strongly connected graphs, it means that no other node M such that $M \equiv_{\text{tree}} N$ have been encountered before. The problem is that we have a partial key semantics on these graphs, and $\equiv_{\text{tree}} \subset \equiv_{\text{pk}}$, so we could have $M \not\equiv_{\text{tree}} N$ but $M \equiv_{\text{pk}} N$ in effect representing the same tree. Because $M \not\equiv_{\text{tree}} N$, there is a path label p such that $M.p$ and $N.p$ do not have the same label, k_M and k_N. But as N and M represent the same tree, k_M and k_N must have the same name, so their only possible difference is in the partial function. It means there is an i such that one of the keys is defined on i and not the other key (if both of them were defined on i, their value would be the same on i, as the nodes in partial keys are unique representations). By construction, the nodes M and N are in strongly connected graphs. So if one of the keys is not defined on i, there is a q such that $M.piq = M$ or $N.piq = N$. If t is the tree represented by both nodes, it means that $t_{[piq]} = t$. Suppose k_M is defined on i, then there is a node reachable from $k_M(i)$ which represents the same tree as M, and as such it would have been found by `shareWithDone`. So the graph defined by M would never have gone beyond the step 1 of `representCycle`. It means that another representative is stored for the cycle (we go on like this until we find one which is equivalent to N, which means that the test step 3 could not have been false). If k_N is defined on i, by the same argument, we could not have been beyond the step 1, and so no new node is created.

If no node equivalent to N has been encountered, it is the same for every other node M in the graph represented by N. It is due to the strong connectivity of the graph which implies that if M has already been encountered, N has already been encountered.

5. Complexity Issues

Algorithms on shared trees can be more difficult than standard algorithms on trees, because we must keep the uniqueness of the representation, and for efficiency, we must do it incrementally. Comparing complexities of algorithms on the two representations (the naive and the sharing ones) is difficult, though. The complexity is measured with respect to the size of the inputs of the algorithms, which can be reduced to the number of nodes of the inputs in our case. In the case of shared regular trees, the number of nodes is exactly the number of distinct subtrees of the tree, but when the tree is not shared, the number of nodes can be of any value greater than the number of distinct subtrees. In the sequel, we denote by n this number of nodes, but we must keep in mind that this n can be much bigger in the case of non-shared trees.

Table I: Summary of worst case time complexities

	sharing representation	naive representation
testing $t_1 = t_2$	$\mathcal{O}(1)$	$\mathcal{O}\left((n_1+n_2)\log(n_1+n_2)\right)$
testing t_1 subtree of t_2	$\mathcal{O}(n_2)$	$\mathcal{O}\left((n_1+n_2)\log(n_1+n_2)\right)$
building $t_{[p]}$	$\mathcal{O}(\lvert p\rvert)$	$\mathcal{O}(\lvert p\rvert)$
root construction	$\mathcal{O}(1)$	$\mathcal{O}(1)$
recursive construction	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

The basic property of shared trees is the uniqueness of the representation. Thus, testing tree equality is really immediate: we just compare the memory location of the root. In the classical case, the best method uses a partitioning algorithm. Another case where we can avoid such a computation with shared trees is testing if a tree is a subtree of another one. In the shared case, we just have to compare the root of the first tree with all the nodes of the second one. Not only is it linear, but the second tree is very likely to have very less nodes in the shared case than in the classical representation.

When building finite trees, we need only one operation, which we call root construction: we give a label f and the nodes $(N_i)_{i<n}$, and we build

$$\begin{tikzcd} & f \arrow[dl] \arrow[dr] & \\ N_0 & \cdots & N_{n-1} \end{tikzcd}$$

. Such an operation is constant time in the naive representation and in the sharing representation for finite trees (assuming hashing is constant time, see Knuth [1973] and Cai and Paige [1994] for a discussion). It is indeed also constant time for infinite trees, but this operation does not suffice to build every regular tree. We need also some loop building mechanism. We call this second operation recursive construction. Considering a tree t and a label x, it consists in replacing every edge going to x by an edge to the root, and then apply `representCycle` to maintain the uniqueness of the representation. This can be done efficiently by adapting the `representation` algorithm. Concerning the complexity of this algorithm, it seems that the prevailing operation is the final (and unique) call to `share`, which is applied on the smallest possible subgraph, but in the worst case, the quadratic complexity of `shareWithDone` will take precedence.

As we announced earlier, many other operations can be adapted to shared trees while preserving the uniqueness of the representation by derivation from the `representation` algorithm. For example, tree substitution is obtained by using an initial `encountered` containing the substitution, plus a few optimizations. We believe the reader will easily create his own adaptations to fit his needs.

The summary of Table I suggests that if we are to perform equality testing, it can be beneficial to perform sharing during the calculus. What we show here is worst case complexity, though, and the difficult cases are quite pathological, and thanks to some simple optimizations, they are quite rare. The situation is very similar to the complexity of operations on BDDs [Bryant 1986] compared to the operations on boolean formulas. The size of the formula representing a given boolean function is unbounded, but the basic operations, like conjunctions, are linear in the size of one of the formulas whereas they are quadratic for the BDDs. Nevertheless, in practice BDDs are far more efficient.

6. Application: Set-Based Analysis

We propose to use these techniques to improve the representation of sets of trees. The expressive power of this improved representation is exactly what is needed in set-based analysis [Heintze 1992], where sets of trees are approximated by ignoring the dependencies between variables (an idea which was already present in Reynolds [1969] and Jones and Muchnick [1979]). This approximation is called the "cartesian approximation".

Although this approximation can be quite crude, it has been introduced to obtain more tractable analyzes. The problem is that the representation used in tools based on set-based analysis is not always optimal. For example, in Heintze and Jaffar [1990], a new variable is introduced for each union, and intersection requires an exponential number of variables. For better results, some authors propose the use of tree automata (Devienne *et al.* [1997] and Charatonik and Podelski [1998]).

6.1 Tree Skeletons

In order to represent the sets of set-based analysis as trees, we propose the use of a new label in the trees, which will be treated in a special way. This label, which we call a choice label corresponds to a possible union in the interpretation of the infinite tree. We denote this label $\bigcirc$. We call the infinite trees with this extra label tree skeletons (see Fig. 8 for examples of tree skeletons). The set of trees represented by a tree skeleton is defined by:

$$\mathrm{Set}\left(\begin{array}{c} f \\ \swarrow\ \searrow \\ t_0 \cdots t_{n-1}\end{array}\right) \overset{\text{def}}{=} \left\{ \begin{array}{c} f \\ \swarrow\ \searrow \\ u_0 \cdots u_{n-1}\end{array} \;\middle|\; \forall i < n, u_i \in \mathrm{Set}\,(t_i) \right\}$$

$$\mathrm{Set}\left(\begin{array}{c} \bigcirc \\ \swarrow\ \searrow \\ t_0 \cdots\cdots t_{n-1}\end{array}\right) \overset{\text{def}}{=} \bigcup_{i<n} \mathrm{Set}\,(t_i)$$

As this is a fixpoint equation, we need to define what fixpoint we mean. In this article, Set is defined as the least fixpoint of this set of equations. The ordering is the pointwise ordering of the inclusion of the images. If we wanted to include infinite trees (as in Charatonik and Podelski [1998]), we would take the greatest fixpoint.

In order to have a unique representation of the sets of trees (and so keep the constant time equality testing and memoizing properties), we make some

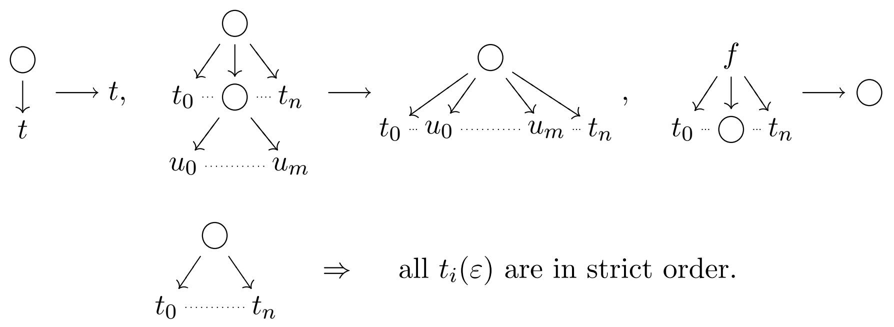


Fig. 7: Rules to obtain a valid tree skeleton

restrictions on what infinite trees are considered valid tree skeletons. First we eliminate unnecessary choices: if a choice node has only one child, it is replaced by its child. If a choice node is the child of a choice node, it is replaced by its children. We perform the cartesian approximation: if two children of a choice node have the same label, they are merged (replaced by their cartesian upper approximation). Finally, the children of a choice node are ordered according to their labels. See the summary of Fig. 7.

6.2 Tree Automata and Tree Skeletons

Because the cartesian approximation eliminates any dependencies between children of a tree, we can use deterministic top-down tree automata in set-based analysis. The idea we use here is that deterministic top-down tree automata can be seen as graphs, where the only properties that matter are path properties, and so they can be represented efficiently as special regular infinite tree: tree skeletons.

A deterministic top-down tree automaton (Thatcher and Wright [1968], and Gécseg and Steinby [1984]) is a tuple (Q, I, δ, F) where Q is a finite set of states, $I \in Q$ is the initial state, $F \subset Q$ is a set of final states, and $\delta : A \times Q \rightarrow Q \times \ldots \times Q$ is the transition function which takes a label in A and a state, and returns a sequence of states (as many as the arity of the label). The corresponding graph G is such that $G^N = Q$, $G^E = \{(q, q', a_i) \mid \delta(a, q) = (\ldots, q', \ldots)$ and q' in i^{th} position $\}$. This connection means that we can represent the sets used in set-based analysis without any variable name in the representation, and in a shared way.

Any deterministic top-down tree automaton can be represented by a valid tree skeleton. Consider an automaton (Q, I, δ, F). We first build the infinite tree labeled by Q and A, such that the root is labeled by I, the children of a given state q are the different a such that $\delta(q, a)$ is defined, and the children of such an a are the $\delta(q, a)$. This tree is regular because there is at most one subtree labeled by a given $q \in Q$, and at most $|Q|$ subtrees labeled by

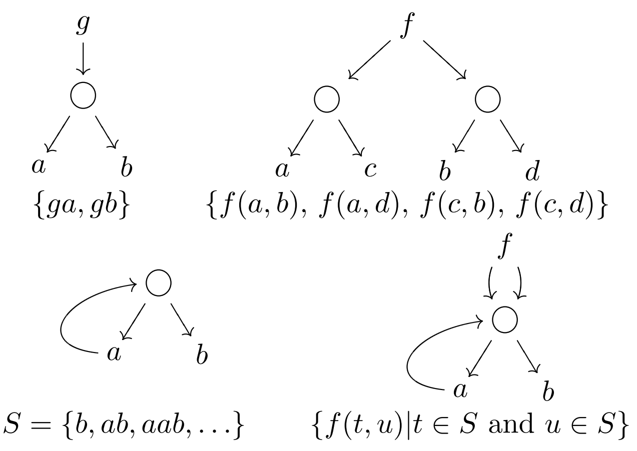

g
a
b
$\{ga, gb\}$
f
a
c
b
d
$\{f(a,b),\ f(a,d),\ f(c,b),\ f(c,d)\}$
a
b
$S = \{b, ab, aab, \ldots\}$
f
a
b
$\{f(t,u) | t \in S \text{ and } u \in S\}$

Fig. 8: Examples of tree skeletons

a given $a \in A$. The second step consists in removing every label of arity 0 which does not come from a state in F, and in replacing every state by $\bigcirc$. Then we derive the valid tree skeleton.

6.3 Using Tree Skeletons in Analysis

Manipulation of tree skeletons uses basic algorithms on shared infinite regular trees. Once we can keep the maximal sharing property, it is easy to keep track of the other rules for tree skeletons. See Mauborgne [1999, chapter 6] for an extensive description of the algorithms manipulating tree skeletons.

Then tree skeletons can be used everywhere we consider a set of trees in the analysis. It can replace some of the tree automata of Devienne *et al.* [1997] (if we keep the original restrictions of set-based analysis), or the tree grammars of Liu [1998], as the approximation on union corresponds indeed to cartesian approximation. In practice, all you have to do is use a toolbox implementing the operations on tree skeletons. Such a toolbox will soon be available at `http://www.di.ens.fr/~mauborgn/`.

7. Conclusion

While trying to improve the representation of sets of trees in set-based analysis, we presented generic algorithms to manipulate efficiently any structure encoded as infinite regular trees. These algorithms allow a very compact representation of such structures and a constant time equality testing. One of their advantages is their incrementality which allows their use on dynamic structures. The complexity analysis cannot describe the potential benefit of this new representation, but it suggests the same gain as for Binary Decision Diagrams which use similar techniques.

We also described a new way of representing sets of trees using infinite regular trees. This new representation is sharing, incremental and unique. Current work includes the integration of the representation in an actual analyzer to show experimentally its benefits.

Acknowledgements

I would like to thank Neil Jones for his useful comments and his faithful support.

Appendix A. Ancillary Algorithms

Appendix A.1 The Partitioning Algorithm

This algorithm is a simplified and specialized version of Cardon and Crochemore [1982]. It modifies the graph reachable from the node G into another graph reachable from G, representing the same tree, and with minimal number of nodes.

```
share($G$):
  Blocks is a function which associates an empty set with every integer
  BCount $\leftarrow$ 0
  $D \leftarrow \emptyset$
  initiateBlocks($G$)
  MaxArity is the maximal arity of the labels in the graph
  for $i \leftarrow 0$ to MaxArity $- 1$ do
    $a \leftarrow$ the block with the greatest number of incoming edges labeled by $i$
    ToTest($i$) $\leftarrow$ [BCount]$\setminus\{a\}$
  while $\exists\, i$ such that ToTest($i$) $\neq \emptyset$ do
    take a $a$ out of ToTest($i$)
    $k \leftarrow$ BCount $- 1$
    for $b \leftarrow 0$ to $k$ do
      Blocks(BCount) $\leftarrow \{N \in$ Blocks($b$) $\mid \exists M \in$ Blocks($a$), $N \rightarrow^i M\}$
      if Blocks(BCount) $\neq \emptyset$ and Blocks(BCount) $\neq$ Blocks($b$) then
        for $j \leftarrow 0$ to MaxArity $- 1$ do
          if the number of nodes leading (labeled by $j$) to the block $b$ is
               greater than the number of nodes leading to the block BCount
               and $b \notin$ ToTest($j$) then add $b$ to ToTest($j$)
          else add BCount to ToTest($j$)
        BCount $\leftarrow$ BCount $+ 1$
  for $a \leftarrow 0$ to BCount $- 1$ do
    if |Blocks($a$)| $> 1$ then
      if $G \in$ Blocks($a$) then $N \leftarrow G$
      else $N$ is any node in Blocks($a$)
      $\forall M \in$ Blocks($a$), $M \neq N$ do
```

```
        every edge $O \to^i M$ is replaced by $O \to^i N$
        remove $M$ from $G^N$

initiateBlocks($N$):
  if $N \notin D$ then
    add $N$ to $D$
    $N$ is labeled by $\begin{smallmatrix} & f & \\ \swarrow & & \searrow \\ N_0 & \cdots & N_{n-1} \end{smallmatrix}$
    if $\exists\, i <$ BCount such that Blocks($i$) contains a node labeled by $f$ then
      add $N$ to Blocks($i$)
    else
      Blocks(BCount) $\leftarrow \{N\}$
      BCount $\leftarrow$ BCount $+ 1$
    for $i \leftarrow 0$ to $n-1$ do initiateBlocks($N_i$)
```

The algorithm first creates a partition (represented by `Blocks`) according to the labels of the nodes, and then progressively refines it until we get the coarsest congruence which refines the initial partition.

Appendix A.2 Keys of Strongly Connected Graphs

This algorithm returns a finite tree identifying uniquely a node in a cyclic graph.

```
treeKey($N$):
  $S$ is a partial function from $G^N$ to $\mathbb{N}^*$ and $dom(S) = \emptyset$
  return (treeKeyRec($\varepsilon, N$))

treeKeyRec($p, N$):
  if $N \in dom(S)$ then return $(S(N))$
  add $N \to p$ to $S$
  $N$ is labeled by $\begin{smallmatrix} & f & \\ \swarrow & & \searrow \\ N_0 & \cdots & N_{n-1} \end{smallmatrix}$
  for $i \leftarrow 0$ to $n-1$ do $t_i \leftarrow$ treeKeyRec($p.i, N_i$)
  return $\left(\begin{smallmatrix} & f & \\ \swarrow & & \searrow \\ t_0 & \cdots & t_{n-1} \end{smallmatrix}\right)$
```

Appendix A.3 Finding Equivalent Nodes in Partial Keys

This program uses two sets of nodes, which are supposed to be empty at the initial call: the set `encount` of encountered nodes of the graph with partial keys we have to test, and the set `done` of encountered nodes in the graphs reachable through the partial keys. The dictionary D_C contains the mapping between the nodes of the graph we have to test and the equivalent nodes reachable through partial keys. The result of `shareWithDone`(N) is N if no other node is equivalent, and the equivalent node otherwise.

```
shareWithDone($N$)
  if $N \in$ encount then return ($N$)
  add $N$ to encount
  $N$ is labeled by $\begin{array}{c} k \\ \swarrow \ \searrow \\ N_0 \cdots N_{n-1} \end{array}$
  $\forall i \in dom(k)$ do if tryShare($N, k(i), i, k(i)$) then return ($D_C(N)$)
  for $i \leftarrow 0$ to $n-1$ do if shareWithDone($N_i$) $\neq N_i$ then return ($D_C(N)$)
  return ($N$)

tryShare($N, M, i, O$)
  if $M \in$ done then return (false)
  add $M$ to done
  if $N$ and $M$ have the same label and $M.i = O$ then
    $D_C$ is an empty node dictionary
    if compareWithDone($N, M$) then return (true)
  $M$ is labeled by $\begin{array}{c} f \\ \swarrow \ \searrow \\ M_0 \cdots M_{n-1} \end{array}$
  return $\left(\bigvee_{j<n}$ tryShare($N, M_j, i, O$)$\right)$

compareWithDone($N, M$)
  if $N \in dom(D_C)$ then
    if $D_C(N) = M$ then return (true) else return (false)
  $N$ is labeled by $\begin{array}{c} f \\ \swarrow \ \searrow \\ N_0 \cdots N_{n-1} \end{array}$ by extending the partial key $k$ labeling $N$
  $M$ is labeled by $\begin{array}{c} g \\ \swarrow \ \searrow \\ M_0 \cdots M_{m-1} \end{array}$
  if $f = g$ then
  | add $N \to M$ to $D_C$
  | for $i \leftarrow 0$ to $n-1$ do
  |   if $i \in dom(k)$ then if $N_i \neq M_i$ then return (false)
  |   else if compareWithDone($N_i, M_i$) = false then return (false)
  | return (true)
  else return (false)
```

References

Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. 1983. *Data Structures and Algorithms*. Addison-Wesley.

Bryant, Randal E. 1986. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers C-35*, (August), 677–691.

Cai, Jiazhen and Paige, Robert. 1994. Using multiset discrimination to solve language processing without hashing. *Theoretical Computer Science* **?**, , ?–?

Cardon, A. and Crochemore, M. 1982. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science 19*, 85–98.

Charatonik, Witold and Podelski, Andreas. 1998. Co-definite Set Constraints. In *9th International Conference on Rewriting Techniques and Applications*, Volume 1379 of *Lecture Notes in Computer Science*. Springer-Verlag, 211–225.

Colmerauer, Alain. 1982. PROLOG and Infinite Trees. In *Logic Programming*, Volume 16 of *APIC Studies in Data Processing*. Academic Press, 231–251.

Devienne, P., Talbot, JM., and Tison, Sophie. 1997. Solving classes of set constraints with tree automata. In *3th International Conference on Principles and Practice of Constraint Programming*, Volume 1330 of *Lecture Notes in Computer Science*. Springer-Verlag, 62–76.

Gécseg, F. and Steinby, M. 1984. *Tree Automata*. Akadémia Kiadó.

Heintze, Nevin. 1992. *Set Based Program Analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University.

Heintze, Nevin. 1994. Set-Based Analysis of ML Programs. In *LFP '94*.

Heintze, Nevin and Jaffar, Joxan. 1990. A Finite Presentation Theorem for Approximating Logic Programs. In *ACM Symposium on Principles of Programming Languages (POPL'90)*, 197–209.

Hopcroft, John. 1971. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*. Academic Press, 189–196.

Jones, N. D. and Muchnick, S. S. 1979. Flow analysis and optimization of LISP-like structures. In *6th POPL*. ACM Press, 244–256.

Knuth, Donald E. 1973. *Sorting and Searching*. Volume 3 of *The Art of Computer Programming*. Addison-Wesley.

Liu, Yanhong A. 1998. Dependence Analysis for Recursive Data. In *IEEE International Conference on Computer Languages*, 206–215.

Mauborgne, Laurent. 1999. Binary Decision Graphs. In *Static Analyis Symposium (SAS'99)*, Volume 1694 of *Lecture Notes in Computer Science*. Springer-Verlag, 101–116.

Mauborgne, Laurent. 1999. *Representation of Sets of Trees for Abstract Interpretation*. PhD thesis, École Polytechnique.

Michie, D. 1968. "Memo" functions and machine learning. *Nature 218*, (April), 19–22.

Reynolds, J. 1969. Automatic computation of data set definitions. In *Information Processing '68*. Elsevier Science Publisher, 456–461.

Thatcher, J. W. and Wright, J. B. 1968. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory 2*, 57–82.