Symbolic Data Structure for sets of $k$-uples of integers

Pierre Ganty$^1$, Cédric Meuter$^1$, Laurent Van Begin$^1$, Gabriel Kalyon$^1$, Jean-François Raskin$^1$, and Giorgio Delzanno$^2$

$^1$ Département d’Informatique,
Université Libre de Bruxelles
$^2$ Dipartimento di Informatica e Scienze dell’Informazione,
Università degli Studi di Genova

Abstract. In this document we present a new symbolic data structure dedicated to the manipulation of (possibly infinite) sets of $k$-uples over integers, initially introduced in [Gan02]. This new data structure called Interval Sharing Tree (IST), is based on sharing trees [ZL95] where each node is labelled with an interval of integers. We present symbolic algorithm on IST for standard set operations and also introduce some specific operation that can be useful in the context of model-checking.

1 Introduction

Many systems like multi-threaded programs and communication protocols are very complex to devise, hence error prone. For almost three decades a lot of research has been done in order to formalize and prove or refute properties on such systems. Given a formal description $M$ of the system, we apply model checking methods [CGP99] to verify automatically that some properties hold on $M$. Those automatic methods manipulate (possibly) infinite sets of configurations of $M$. In model checking one problem we face off is the state explosion problem. This problem corresponds to a blow up in the size of the sets of configurations being manipulated. One way to overcome that problem is to use a compact symbolic representation of those sets. In our context assuming that those configurations are $k$-uples over integers, we define Interval Sharing Tree (IST) [Gan02] which is a symbolic representation of sets of $k$-uples. ISTs are directed acyclic graph-based data structures, where nodes are labelled with intervals of integers. This symbolic data structure benefits from several advantages. First, algorithms can be defined to manipulate ISTs symbolically (i.e. without exploring all path of the graph). Second, the size of an IST $I$ may be exponentially smaller than the size of the set of $k$-uples it represent. Finally, an IST $I$ can be (semantically) non-deterministic whereas other approaches facing to the state explosion problem traditionally use deterministic data structures (e.g. sharing trees [ZL95], BDD [CGP99], ...), thus allowing $I$ to be more compact.

The remainder of this document is structured as follows. First, in sec. 2, we introduce interval sharing trees and their semantics. We follow up, in sec. 3,
by presenting symbolic algorithms for manipulating this data structure. Finally, in sec. 4, we conclude by presenting a summary of all operations and their complexity.

2 Interval Sharing Tree

In the following, \( \mathbb{Z} \) denotes the set of integers and \( \mathbb{Z}^{\infty} = \mathbb{Z} \cup \{-\infty, +\infty\} \). The classical order over integer is extended to \( \mathbb{Z}^{\infty} \) in the usual way: \( \forall c \in \mathbb{Z} : -\infty < c < +\infty \). An interval is a pair \( (a, b) \in \mathbb{Z}^{\infty} \times \mathbb{Z}^{\infty} \) such that \( a \leq b \), noted \([a, b] \).

As usual, an interval \( (a, b) \) denotes the set \( \{c \in \mathbb{Z} \mid a \leq c \leq b\} \). Hence, \( c \in [a, b] \) means that \( c \) is in the set corresponding to \([a, b]\). We note \( \| \) the set of all intervals. Moreover, we note \( (a, b) = [a, b] \setminus \{a\} \) and \([a, b) = [a, b] \setminus \{b\} \).

**Definition 1 (Interval Sharing Tree).** An interval sharing tree (IST in short) \( I \), is a labelled directed rooted acyclic graph \( \langle N, \iota, \text{succ} \rangle \) where:

- \( N = N^0 \cup N^1 \cup N^2 \cup \ldots \cup N^k \cup N^{k+1} \) is the finite set of nodes, partitioned into \( k + 2 \) disjoint subsets \( N^i \) called layers with \( N^0 = \{\text{root}\} \) and \( N^{k+1} = \{\text{end}\} \).
- \( \iota : N \mapsto \mathbb{I} \cup \{\top, \bot\} \) is the labelling function such that \( \iota(n) = \top \) (resp. \( \bot \)) if and only if \( n = \text{root} \) (resp. \( \text{end} \)).
- \( \text{succ} : N \mapsto 2^N \) is the successor function such that:
  
  (i) \( \text{succ}(\text{end}) = \emptyset \)
  
  (ii) \( \forall i \in [0, k], \forall n \in N^i : \text{succ}(n) \subseteq N^{i+1} \land \text{succ}(n) \neq \emptyset \)
  
  (iii) \( \forall n \in N, \forall n_1, n_2 \in \text{succ}(n) : (n_1 \neq n_2) \Rightarrow (\iota(n_1) \neq \iota(n_2)) \)
  
  (iv) \( \forall i \in [0, k], \forall n_1 \neq n_2 \in N^i : (\iota(n_1) = \iota(n_2)) \Rightarrow (\text{succ}(n_1) \neq \text{succ}(n_2)) \)

In other words, an IST is a directed acyclic graph where the nodes are labelled with intervals except for two special nodes (\text{root} and \text{end}), such that (i) the \text{end} node has no successors, (ii) all nodes from layer \( i \) have their successors in layer \( i + 1 \), (iii) a node cannot have two successors with the same label, (iv) two nodes with the same label in the same layer do not have the same successors. Condition (iii) ensures maximal prefix sharing, i.e. the IST is syntactically deterministic. It does not mean that an IST is \textit{semantically deterministic}, i.e. a node may have two distinct successors labeled with intervals that are not disjoint. Condition (iv) ensures maximal suffix sharing. If a graph satisfies all condition of definition 1 by those two condition, it is called a pre-IST.

A path of an IST is a \( k \)-uple \( \langle n_1, n_2, \ldots, n_k \rangle \) such that \( n_1 \in \text{succ(root)} \), \( \text{end} \in \text{succ}(n_k) \) and \( \forall i \in [1, k] : n_{i+1} \in \text{succ}(n_i) \). Given an IST \( I \), we note \( \text{path}(I) \), the set of path of \( I \). The semantics of an IST \( I \), noted \( [I] \) is the set \( \{ x \in \mathbb{Z}^{k} \mid \exists (n_1, \ldots, n_k) \in \text{path}(I), \forall i \in [1, k] : x_i = \iota(n_i) \} \).

3 Operations on IST

In this section we present the operations on IST. Those operation are symbolic, i.e. they work on the tree instead of the set of \( k \)-uples that are represented.
We start in sec. 3.1 by introducing some preliminary operations that will later on. Then, in sec. 3.2, we introduce basic set operations, i.e. emptiness test, membership test, union, intersection and complement. Finally, in sec. 3.3, we present some operations that are useful in the context of model-checking, namely substitution and downward closure.

3.1 Preliminary operations

Prefix sharing Given a pre-IST $\mathcal{I}$ with $k$ layers, we want to modify $\mathcal{I}$ so that it satisfies condition (iii) of definition 1. The idea is to traverse the sharing tree layer-by-layer and to determinize syntactically the nodes of each layer. This algorithm has a worst-case time complexity of $O(2^{|\mathcal{I}|})$. Note however, that this algorithm returns a pre-IST satisfying condition iii, but not necessarily condition (iv).

Algorithm 1: enforcePrefixSharing($\mathcal{I}, k$)

| data: $\mathcal{I} = (N, \iota, \text{succ})$, $k \in \mathbb{N}$ |
| post: $\forall n \in N, \forall n_1, n_2 \in \text{succ}(n) : (n_1 \neq n_2) \Rightarrow (\iota(n_1) \neq \iota(n_2))$ |

begin

for $i \leftarrow 0$ to $k$ do

forall $n \in N^i$ do

$S \leftarrow \emptyset$

forall $I \in \mathcal{I}$ s.t. $\exists n' \in \text{succ}(n) : \iota(n') = I$

let $n''$ be a new node
$
\text{succ}(n'') \leftarrow \emptyset$, $\iota(n'') \leftarrow I$

forall $n' \in \text{succ}(n)$ s.t. $\iota(n') = I$

\[ \text{succ}(n'') \leftarrow \text{succ}(n'') \cup \text{succ}(n') \]

$N^{i+1} \leftarrow N^{i+1} \cup \{n''\}$

$S \leftarrow S \cup \{n''\}$

$\text{succ}(n) \leftarrow S$

end

Suffix sharing Given a pre-IST $\mathcal{I}$ with $k$ layers, we want to transform $\mathcal{I}$ so that it satisfies condition (iv) of definition 1. The idea is to traverse the sharing tree layer-by-layer (bottom-up). When we encounter two nodes in same layer with the same label and with the same set of sons, we delete one of them (the other one takes its place). Note that this operation preserves condition iii. Therefore, if one wants to enforce both conditions, one must start with enforcePrefixSharing().
Algorithm 2: enforceSuffixSharing(\(I, k\))

**data:** \(I = (N, \iota, \text{succ})\), \(k \in \mathbb{N}\)

**post:** \(\forall i \in [0, k], \forall n_1 \neq n_2 \in N^i : (\iota(n_1) = \iota(n_2)) \Rightarrow (\text{succ}(n_1) \neq \text{succ}(n_2))\)

**begin**

\[
\text{for } i \leftarrow k \text{ do until } i = 1 \\
\text{forall } n \in N^i \\
\qquad \text{forall } n' \in N^i \setminus \{n\} \text{ s.t. } \iota(n) = \iota(n') \wedge \iota(n) \in \text{succ}(n) = \text{succ}(n') \text{ do} \\
\qquad \text{forall } n'' \in N^{i-1} \text{ s.t. } n' \in \text{succ}(n'') \text{ do} \\
\qquad\quad \text{\text{succ}(n'') \leftarrow (\text{succ}(n'') \setminus \{n'\}) \cup \{n\} } \]

\[
N^i \leftarrow N^i \setminus \{n'\} \\
\text{end}
\]

\[\text{Algorithm 3: determinize}(I, k)\]

**data:** \(I = (N, \iota, \text{succ})\), \(k \in \mathbb{N}\)

**post:** \(\forall n \in N, \forall n_1 \neq n_2 \in \text{succ}(n) : \iota(n_1) \cap \iota(n_2) = \emptyset\)

**begin**

\[
\text{for } i \leftarrow 0 \text{ to } k \text{ do} \\
\text{forall } n \in N^i \text{ s.t. } \exists n_1 \neq n_2 \in \text{succ}(n) : \iota(n_1) \cap \iota(n_2) \neq \emptyset \text{ do} \\
\text{let } n' \text{ be a new node} \\
\iota(n') \leftarrow \iota(n) \\
\text{forall } n'' \in N \text{ s.t. } n \in \text{succ}(n'') \text{ do} \\
\text{\text{\text{succ}(n'') \leftarrow \text{\text{\text{succ}(n'')} \cup \{n\}}} \}
\]

\[
L \leftarrow \{\iota(n'') | n'' \in \text{succ}(n)\} \\
\text{let } L' \in 2^I \text{ be the smallest partition of } \bigcup_{I \in L} I \] 

\[
\text{forall } I' \in L' \text{ do} \\
\text{let } s \text{ be a new node} \\
\text{forall } t \in \text{succ}(n) \text{ s.t. } I' \subseteq \iota(t) \text{ do} \\
\text{\text{\text{succ}(s) \leftarrow \text{\text{\text{succ}(s)} \cup \text{\text{\text{succ}(t)}}}} \}
\]

\[
\iota(s) \leftarrow I' \\
\text{\text{\text{succ}(s)} \leftarrow \text{\text{\text{succ}(s)} \cup \{s\}}} \\
\text{\text{\text{\text{N}^i+1 \leftarrow \text{\text{\text{\text{N}^i+1} \cup \{s\}}} \}} \\
N^i \leftarrow N^i \cup \{n\} \setminus \{n\} \\
\text{end}
\]

**Determinization** Given an IST \(I\) with \(k\) layers, we want to eliminate from \(I\) the semantical non-determinism that can arise in the successors of a given node while preserving \([I]\). After the determinization of a sharing tree, the intervals defined by the successors of any node must be disjoint. The idea of the algorithm is to traverse the sharing tree layer-by-layer and for each node \(n\), compute a new set of disjoint intervals. Then, we create a node \(s\) for each such interval and we compute the successors of \(s\), i.e. the successors of nodes that belongs to \(\text{succ}(n)\).
and that have a label containing the label of \( s \). This algorithm has a worst-case complexity of \( O(2^{|I|}) \).

### 3.2 Set operations

#### Emptiness

Given an IST \( I \) with \( k \) layers, we want to decide if \( [I] = \emptyset \) holds. The algorithm for that is very simple. Indeed, \( [I] = \emptyset \) if and only if \( \text{succ}(\text{root}) = \emptyset \). This algorithm has a worst-case time complexity of \( O(1) \).

**Algorithm 4: empty(\( I, k \))**

- **data**: \( I = (N, \iota, \text{succ}) \), \( k \in \mathbb{N} \)
- **returns**: \( \text{true} \) if and only if \( [I] = \emptyset \)

```
begin
return \( \text{succ}(\text{root}) = \emptyset \)
end
```

#### Membership

Given an IST \( I \) with \( k \) layers and a \( k \)-uple \( \overline{x} = (x_1, \ldots, x_k) \in \mathbb{Z}^k \), we want to determine if \( \overline{x} \in [I] \) holds. For that, we present an algorithm for that problem adapted from Covering Sharing Trees [Van03,DRV04] that has a worst-case time complexity \( O(|I|) \). The algorithm is based on a layer-by-layer traversal of the graph. We first mark the root node of \( I \). Then for each layer \( i \in [1, k+1] \), we mark all nodes \( n \in N_i \) such that \( x_i \in \iota(n) \) and such that \( n \) has a marked predecessor. At the end of this procedure, \( \overline{x} \in I \) holds if and only if the end node has at least one marked predecessor.

**Algorithm 5: memberOf(\( I, k, \overline{x} \))**

- **data**: \( I = (N, \iota, \text{succ}) \), \( k \in \mathbb{N} \), \( \overline{x} \in \mathbb{Z}^k \)
- **returns**: \( \text{true} \) if and only if \( \overline{x} \in [I] \) holds

```
begin
M ← \{ \text{root} \}
for \( i \leftarrow 1 \) to \( k \) do
    forall \( n \in N^i \) do
        if \( (x_i \in \iota(n)) \land (\exists n' \in M : n \in \text{succ}(n')) \) then
            M ← M \cup \{ n \}
    return (\exists n \in M : \text{end} \in \text{succ}(n))
end
```

#### Intersection

Given two IST \( I_1, I_2 \) with \( k \) layers, we want to compute an IST \( I \) such that \( [I] = [I_1] \cap [I_2] \). The algorithm is adapted from the algorithm computing the synchronized product of finite automata. Each node in this product is a pair of node from both \( I_1 \) and \( I_2 \). In this particular case, the interval
of a node \((n_1, n_2)\) in the product is the intersection of the interval of \(n_1\) and \(n_2\). Unfortunately, this can introduce some prefix and/or suffix redundancy, thus violating conditions (iii) and/or (iv) of definition \(1\). Therefore, we must enforce those two conditions in the algorithm to turn the resulting pre-IST into an IST. For this reason, this algorithm has a worst-case time complexity of \(O(2^{|I_1| \times |I_2|})\).

It was proven in [Van03, thm. 20, 21] that for Covering Sharing Trees, which is a subclass of IST, no polynomial algorithm exists for that problem.

Algorithm 6: intersection\((I_1, I_2, k)\)

`data`: \(I_1 = (N_1, \iota_1, \text{succ}_1), I_2 = (N_2, \iota_2, \text{succ}_2), k \in \mathbb{N}\)

`returns`: \(I\) s.t. \([I] = [I_1] \cap [I_2]\)

begin
\[
I \leftarrow \langle N^0 \cup \ldots \cup N^{k+1}, \iota, \text{succ} \rangle
\]
for \(i \leftarrow 0\) to \(k\) do
\[
N^i \leftarrow \emptyset
\]
\[
N^{k+1} \leftarrow \{(\text{end}_1, \text{end}_2)\}
\]
\[
\iota(\text{end}_1, \text{end}_2) \leftarrow \bot
\]
for \(i \leftarrow k\) downto \(1\) do
forall \((n_1, n_2) \in N^k \times N^k\) s.t. \(\iota_1(n_1) \cap \iota_2(n_2) \neq \emptyset\) do
\[
\iota(\langle n_1, n_2 \rangle) \leftarrow \iota_1(n_1) \cap \iota_2(n_2)
\]
\[
\text{succ}(\langle n_1, n_2 \rangle) \leftarrow N^{k+1} \cap (\text{succ}_1(n_1) \times \text{succ}_2(n_2))
\]
if \(\text{succ}(\langle n_1, n_2 \rangle) \neq \emptyset\) then
\[
N^k \leftarrow N^k \cup \{(n_1, n_2)\}
\]
\[
N^0 \leftarrow \{(\text{root}_1, \text{root}_2)\}
\]
\[
\iota(\langle \text{root}_1, \text{root}_2 \rangle) \leftarrow \top
\]
\[
\text{succ}(\langle \text{root}_1, \text{root}_2 \rangle) \leftarrow N^1
\]
enforcePrefixSharing\((I, k)\)
enforceSuffixSharing\((I, k)\)
return \(I\)
end

**Union** Given two IST \(I_1, I_2\) with \(k\) layers, we want to compute an IST \(I\) such that \([I] = [I_1] \cup [I_2]\). This can be done using the algorithm to compute the union of two sharing trees defined in [ZL95]. This algorithm uses two recursive functions `recUnion()` and `recCopy()`. The function `recUnion()` computes the union of a sub-tree of \(I_1\) rooted in \(n_1\) and a sub-tree of \(I_2\) rooted in \(n_2\). The roots of these sub-trees must have the same labels. The function `recCopy()` copies a sub-tree of \(I_1\) or \(I_2\) in \(I\). Those two functions make use of a hashing table \(T\) to remember previous computations. When the union of two sub-trees rooted respectively in \(n_1\) and \(n_2\) is computed and added in \(I\), the resulting node is stored in \(T\), indexed by the pair \((n_1, n_2)\). Similarly, when a sub-tree rooted in \(n\) is copied in \(I\), the copied node is stored in \(T\), indexed by the pair \((n, n)\). This is essential in order treat every pair of nodes only once, thus allowing a polynomial...
complexity. Note that condition iv does not necessarily holds on the result and must therefore be enforced. This algorithm has a worst-case time complexity of $O((|I_1| + |I_2|)^3)$.

Algorithm 7: \textit{recUnion}(n_1, n_2, i)

\begin{algorithm}
begin
\begin{algorithmic}
\If{$T(n_1, n_2)$ is defined}
\State $n \leftarrow T(n_1, n_2)$
\Else
\State let $n$ be a new node
\State $\iota(n) \leftarrow \iota_1(n_1)$
\State $N^i \leftarrow N^i \cup \{n\}$
\ForAll{$n'_1 \in \text{succ}_1(n_1)$}
\If{$\exists n'_2 \in \text{succ}_2(n_2)$ s.t. $\iota_1(n'_1) = \iota_2(n'_2)$}
\State $\text{succ}(n) \leftarrow \text{succ}(n) \cup \{\text{recUnion}(n'_1, n'_2, i + 1)\}$
\Else
\State $\text{succ}(n) \leftarrow \text{succ}(n) \cup \{\text{recCopy}(n'_1, 1, i + 1)\}$
\EndIf
\EndFor
\State $T(n_1, n_2) \leftarrow n$
\Return $n$
\EndIf
\end{algorithmic}
\end{algorithm}

Algorithm 8: \textit{recCopy}(n, x, i)

\begin{algorithm}
begin
\begin{algorithmic}
\If{$T(n, n)$ is defined}
\State $n' \leftarrow T(n, n)$
\Else
\State let $n'$ be a new node
\State $\iota(n') \leftarrow \iota_2(n)$
\State $N_i \leftarrow N_i \cup \{n'\}$
\ForAll{$n'' \in \text{succ}_2(n)$}
\State $\text{succ}(n') \leftarrow \text{succ}(n') \cup \{\text{recCopy}(n'', x, i + 1)\}$
\EndFor
\State $T(n, n) \leftarrow n'$
\Return $n'$
\EndIf
\end{algorithmic}
\end{algorithm}
Algorithm 9: union($\mathcal{I}_1, \mathcal{I}_2, k$)

data: $\mathcal{I}_1 = \langle N_1, \iota_1, \text{succ}_1 \rangle$, $\mathcal{I}_2 = \langle N_2, \iota_2, \text{succ}_2 \rangle$, $k \in \mathbb{N}$
returns: $\mathcal{I}$ s.t. $[\mathcal{I}] = [\mathcal{I}_1] \cup [\mathcal{I}_2]$

begin
  $\mathcal{I} \leftarrow \langle N^0 \cup \ldots \cup N^{k+1}, \iota, \text{succ} \rangle$
  forall $i \leftarrow 0$ to $k$ do
    $N^i \leftarrow \emptyset$
    recUnion(root$_1$, root$_2$, 0)
    enforceSuffixSharing($\mathcal{I}$, $k$)
  return $\mathcal{I}$
end

Algorithm 10: complement($\mathcal{I}, k$)

data: $\mathcal{I} = \langle N, \iota, \text{succ} \rangle$, $k \in \mathbb{N}$
returns: $\mathcal{I}_1$ such that $[\mathcal{I}_1] = \mathbb{Z}^k \setminus [\mathcal{I}]$

begin
  $\mathcal{I}_1 \leftarrow \mathcal{I}$
  determinize($\mathcal{I}_1$)
  $M \leftarrow N^k$
  for $i \leftarrow 0$ to $k$ do
    forall $n \in N^i_1$ do
      $L \leftarrow \{ \iota_1(n') \mid n' \in \text{succ}_1(n) \}$
      let $L'$ be the smallest partition of $\mathbb{Z} \setminus (\cup_{I \in L} I)$
      forall $I' \in L'$ do
        let $n'$ be a new node
        $\iota_1(n') \leftarrow I'$, $\text{succ}_1(n) \leftarrow \text{succ}_1(n) \cup \{n'\}$
        $N^{i+1} \leftarrow N^{i+1} \cup \{n'\}$
    forall $n \in M$ do
      $\text{succ}_1(n) \leftarrow \emptyset$
   forall $n \in N^{k-1}_1 \setminus M$ do
      $\text{succ}_1(n) \leftarrow \text{end}_1$
  for $i \leftarrow k$ downto 1 do
    forall $n \in N^i$ do
      if $\text{succ}_1(n) = \emptyset$ then
        $N^i \leftarrow N^i \setminus \{n\}$
      forall $n' \in N^i_1$ s.t. $n \in \text{succ}_1(n')$ do
        $\text{succ}_1(n') \leftarrow \text{succ}_1(n') \setminus \{n\}$
    enforceSuffixSharing($\mathcal{I}_1, k$)
  return $\mathcal{I}_1$
end
Complement Given an IST $\mathcal{I}$ with $k$ layers, we want to compute an IST $\mathcal{I}_1$ such that $[\mathcal{I}_1] = \mathbb{Z}^k \setminus [\mathcal{I}]$. First, we take a copy $\mathcal{I}_1$ of $\mathcal{I}$, and determinize it. Then, for each node $n$ of $\mathcal{I}_1$, we complement the set of intervals defined by the successors of $n$ to obtain a partition of $\mathbb{Z}$. We create a new node for each of these new intervals and add these new nodes to $\text{succ}(n)$. In fact, these new nodes allow to consider the $k$-uples that do not belong to $[\mathcal{I}]$. To delete the $k$-uples that belong to $[\mathcal{I}]$, we delete the edges between the nodes of the layer $N_k^1$ that were copied from $\mathcal{I}_1$ and the node $\text{end}_1$. Finally, we delete from the IST all path not ending in $\text{end}_1$.

3.3 Dedicated operations

Substitution Given an IST $\mathcal{I}$ with $k$ layers, we want to compute an IST $\mathcal{I}_1$ where a certain variable $x_i$ is replaced by $x_i + d$ with $d \in \mathbb{Z}$. Formally, we want that $[\mathcal{I}_1] = [\mathcal{I}]_{x_i + d/x_i} = \{(x_1, \ldots, x_i + d, \ldots, x_k) \mid \pi \in \mathcal{I}\}$. This operation can be done very simply, by shifting all interval of layer $i$. Note that this transformation does not violate condition (iv) of (iii).

\begin{algorithm}
\textbf{Algorithm 11:} substitution($\mathcal{I}, k, i, d$)
\begin{tabular}{l}
\textbf{data} : $\mathcal{I} = \langle N, \iota, \text{succ} \rangle$, $k \in \mathbb{N}$, $i \in [1, k]$, $d \in \mathbb{Z}$
\textbf{returns:} $\mathcal{I}_1$ s.t. $[\mathcal{I}_1] = [\mathcal{I}]_{x_i + d/x_i}$
\textbf{begin}
$\mathcal{I}_1 \leftarrow \langle N, \iota_1, \text{succ} \rangle$
\textbf{forall} $n \in N \setminus N^i$ \textbf{do}
$\iota_1(n) \leftarrow \iota(n)$
\textbf{forall} $n \in N^i$ \textbf{do}
$\langle l, u \rangle \leftarrow \iota(n)$
$\iota_1(n) \leftarrow \langle l + d, u + d \rangle$
\textbf{return} $\mathcal{I}_1$
\end{tabular}
\end{algorithm}

Downward Closure Given a subset $S \subseteq \mathbb{Z}^k$, the downward closure of $S$, noted $\downarrow S$ is the set with $k$-uple smaller than a $k$-uple of $S$. Formally, $\downarrow S = \{ \pi \in \mathbb{Z}^k \mid \exists \pi' \in S \forall i \in [1, k] : x_i \leq x_i' \}$. Hence, given an IST $\mathcal{I}$ with $k$ layers, we want to compute an IST $\mathcal{I}_1$ such that $[\mathcal{I}_1] = \downarrow [\mathcal{I}]$. The algorithm works as follows. The left bound of all nodes is set to $-\infty$. This transformation can introduce some prefix and/or suffix redundancy, thus violating conditions (iii) and/or (iv) of definition 1. Therefore, we must enforce those two conditions in the algorithm. The algorithm has a worst-time complexity $O(2^{\mid \mathcal{I} \mid})$. 
<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>enforcePrefixSharing($I$, $k$)</td>
<td>enforce condition (iii) on $I$</td>
<td>$O(2^{</td>
</tr>
<tr>
<td>determineize($I$, $k$)</td>
<td>semantically determine $I$</td>
<td>$O(</td>
</tr>
<tr>
<td>empty($I$, $k$)</td>
<td>test if $[I] = \emptyset$ holds</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>memberOf($I$, $k$, $x$)</td>
<td>test if $x \in [I]$ holds</td>
<td>$O(</td>
</tr>
<tr>
<td>union($I_1$, $I_2$, $k$)</td>
<td>compute $I$ s.t. $[I] = [I_1] \cup [I_2]$</td>
<td>$O(</td>
</tr>
<tr>
<td>intersection($I_1$, $I_2$, $k$)</td>
<td>compute $I$ s.t. $[I] = [I_1] \cap [I_2]$</td>
<td>$O(</td>
</tr>
<tr>
<td>substitution($I$, $k$, $i$, $d$)</td>
<td>computes $I_1$ s.t. $[I_1] = Z_{i+d/</td>
<td>I_1</td>
</tr>
<tr>
<td>complement($I$, $k$)</td>
<td>compute $I$ s.t. $[I_1] = Z \setminus [I]$</td>
<td>$O(2^{</td>
</tr>
<tr>
<td>downwardClosure($I$, $k$)</td>
<td>compute $I_1$ s.t. $[I_1] = \downarrow [I]$</td>
<td>$O(2^{</td>
</tr>
</tbody>
</table>

Table 1. Operations on IST and their complexity

Algorithm 12: downwardClosure($I$, $k$)

```plaintext
data : $I = \langle N, \iota, \text{succ} \rangle$, $k \in \mathbb{N}$
returns: $I_1$ s.t. $[I_1] = [I]$ begin
    forall $n \in N$ do
        $\langle l, u \rangle \leftarrow \iota(n)$
        $\iota_1(n) \leftarrow (-\infty, u)$
        $I_1 \leftarrow \langle N, \iota_1, \text{succ} \rangle$
        enforcePrefixSharing($I_1$)
        enforceSuffixSharing($I_1$)
    return $I_1$
end```

4 Conclusion

In this paper, we defined a symbolic data structure for sets of $k$-uples of integers, called Interval Sharing Trees. We presented algorithms in order to manipulate this data structure symbolically. Those operations include traditional set operations as well as specific operation dedicated to model-checking, and are summarized, along with complexity in tab. 1.

References


