# APPENDIX A. FORMAL DEFINITION OF P-TREE REPRESENTATIONS

## A.1. P-tree Definition

This appendix gives formal definitions of P-tree representations that are used in the implementation sections of Chapter 3. For a definition of the notation, please refer to [1]. Note that the respective notation is only used in this appendix. The thesis otherwise uses conventional notation. We will use f.x for function application, which is conventionally expressed as f(x), and (Op i: $i \geq i_0 \wedge i \leq i_1$ : f.x) for quantified expressions for which we otherwise use $\mathop{Op}_{i=i_0}^{i_1} f(x)$.

A P-tree can be defined in a recursive fashion. The simplest definition lets each child hold the purity information of its own node, i.e., one Boolean variable that identifies it as a "pure 1" node, p, and one that identifies it as a "pure 0", z, node. The node holds its children as an array of P-tree. Note that the existence of child information together with either p or z could be used to define the tree uniquely. Both variables will be needed for the modified representations that are derived later. The following definition assumes a Peano truth tree, i.e., a tree that does not hold count information. Implementations commonly do hold count information for performance reasons. This information is redundant and can be re-derived by the function cnt; see the following sections.

**Definition 1:** A P-tree of fan-out f and level l is a tuple, t

(f, l, p, z, ch), (1)

where f is of type integer and denotes the fan-out, l is of type integer and stands for the level, p is a Boolean variable that is true if the tree is a pure 1 tree, z is a Boolean variable

that is true if the tree is a pure 0 tree, and ch is an array of P-tree with #ch being the length of the array. The following relationships hold:

| | |
|---|---|
| $((\neg p \wedge \neg z \equiv (\#ch \equiv f))$ | a non-pure tree has f children |
| $\vee ((p \vee z) \equiv (\#ch \equiv 0)))$ | a pure tree has no children |
| $\wedge (\neg p \vee \neg z)$ | a tree cannot be both pure 0 and pure 1 |
| $\wedge ((l \equiv 0) \Rightarrow (\#ch \equiv 0))$ | trees at the lowest level must be pure |
| $\wedge (\forall i : 0 \leq i \wedge i < \#ch : ch.i.f \equiv f)$ | the fan-out is constant throughout the tree |
| $\wedge (\forall i : 0 \leq i \wedge i < \#ch : ch.i.l \equiv l-1)$ | child-level smaller by 1 than parent-level |
| $\wedge (\exists i : 0 \leq i \wedge i < \#ch : \neg ch.i.p)$ | children not represented if all are "pure 1" |
| $\wedge (\exists i : 0 \leq i \wedge i < \#ch : \neg ch.i.z)$ | children not represented if all are "pure 0" |

The lowest level, i.e., the level furthest away from the root, is level l=0. This level is sometimes called leaf-level despite the fact that pure trees at any level are leaves.

A particular P-tree is defined by its equivalence to a sequence of binary digits that can be identified with the data sequence (Peano-sequence) that is used to construct the P-tree. The following recursive function, ds, defines a data sequence:

| | | |
|---|---|---|
| ds.t = | <u>if</u> $p \rightarrow s.1.f^l$ | pure sequence of length $f^l$ |
| | $z \rightarrow s.0.f^l$ | |
| | $(\neg p \wedge \neg z) \rightarrow chs.t.0$ | |
| | <u>fi,</u> | |

where s.d.n and chs.t.n will be explained in the following sections. A P-tree, t, is equivalent to a sequence of binary digits, dsreal, if the data sequence produced by ds.t is identical to dsreal. s.d.n defines a sequence of binary digit d through a recurrence relation.

s.d.n = if (n = 0) → ε

(n > 0) → [d] ~ s.d.(n-1)

fi,

where ε is the empty sequence. It can easily be seen why the sequence has to have $f^l$ elements: for a lowest-level (level 0) tree, one digit is represented. This digit is 1 for a pure 1 tree (p is true) and 0 for a pure 0 tree (z is true). For a level-1-tree, the lowest level (level 0) is not represented. This level would have contained f nodes, each of which would have contributed one binary digit. Therefore, f binary digits have to be catenated with the sequence. It can be seen that the number of represented digits is multiplied by f for each level.

For trees that are not pure, children are processed recursively. Non-pure trees have f children that have to be processed in the order of increasing child-index. Note that only pure nodes contribute binary digits. Since all nodes are pure at level 0, each branch of the tree will contribute to the sequence.

chs.t.n = if (n = f) → ε

(n < f) → ds.(t.ch.n) ~ chs.t.(n+1)

fi

The bit count of a node, i.e., the number of "1" bits that are generated for the sequence that is associated with this node, can be derived in an analogous fashion. In a Peano Count Tree, this count will be stored with the node for performance reasons.

cnt.t = if p → $f^l$ pure 1 node has count $f^l$

(¬p ∧ ¬z) → chcnt.t.0

fi

chcnt.t.n = $\underline{\text{if}}$ (n = f) $\rightarrow$ 0

(n < f) $\rightarrow$ cnt.(t.ch.n) + chcnt.t.(n+1)

$\underline{\text{fi}}$

## A.2. AND Operation

One of the most important properties of P-trees lies in their suitability to Boolean operations. We will look at the example of a logical AND operation. The precondition for the AND operation is ($t_1.f = t_2.f \wedge t_1.l = t_2.l$)

$t_{res}$ = and.$t_1.t_2$ = ($f_{res} = f_1$,

$l_{res} = l_1$,

$p_{res} = t_1.p \wedge t_2.p$

$z_{res} = t_1.z \vee t_2.z$

$\vee$ ($\forall$ i : $0 \le i \wedge i < \#ch$ : (AND.($t_1$.(ch.i)).($t_2$.(ch.i))).z),

$ch_{res}$ such that (($\#ch_{res}=0 \wedge z_{res}$)

$\vee$ ($\forall$ i : $0 \le i \wedge i < f$ : $ch_{res}.i$ = AND.($t_1$.(ch.i)).($t_2$.(ch.i))))

Referring to the sequences, $s_1 = ds.t_1$ and $s_2 = ds.t_2$, it can be shown that this AND operation results in a P-tree that is equivalent to a sequence of binary digits, $s_{res} = ds.t_{res}$ with $\#s_{res} = \#s_1 = \#s_2$, that satisfies the condition $s._{res} = ands.s._1.s_2$ with

ands.$s_1.s_2$ = $\underline{\text{if}}$ ($\#s_1 = 0$) $\rightarrow \varepsilon$

($\#s_1 > 0$) $\rightarrow$ using $s_1 = X \sim x$ and $s_2 = Y \sim y$

if ($X=1 \wedge Y=1$) $\rightarrow$ [1] ~ as.x.y

$(X=0 \vee Y=0) \rightarrow [0] \sim as.x.y$

fi

fi

Since the AND operation is symmetric and associative, it can be generalized to sets of P-trees (quantified version).

The P-tree definition (1) is not suitable for an actual implementation. A direct translation into a programming language would represent child P-trees by pointers. The child array would, therefore, take up significantly more storage than the individual Boolean variables that make up the meaningful content of the tree. Furthermore, there is no inherent parallelism in the evaluations that have to be done when ANDing. We will, therefore, look at more efficient representations.

## A.3. Pre-order Sequence Representation

A natural solution to the storage problem may be the creation of a pre-order sequence of the tree where each node is represented by the pair (p: bool, z:bool). Fan-out and level only has to be stored once, together with the sequence

pre.t = if $(p \vee z) \rightarrow [(p, z)]$

$(\neg p \wedge \neg z) \rightarrow [(p, z)] \sim$ chpre.t.0

fi

chpre.t.n= if $(n = f-1) \rightarrow$ pre.(t.ch.(f-1))

$(n < f-1) \rightarrow$ pre.(t.ch.n) ~ chpre.t.(n+1)

fi

Such an implementation is very storage efficient, but performance of the AND operation does not benefit as much from compression as it does for other representations we will discuss. Since there is no way to access a particular node directly, the entire sequence of all trees has to be scanned for each AND, resulting in poor scaling with the number of trees that are being ANDed. Furthermore, we are still not exploiting parallelism in bit-wise operations.

## A.4. Array-sequence Equivalence

The representations that will be discussed in the rest of this appendix will frequently rely on the conversion of a sequence into an array. To this aim, we introduce the concept of array-sequence equivalence.

**Definition 2:** An array, a, is equivalent to a sequence, s, with length #s if

var a: array of T

var s: sequence of T with #s = #a

; ($\forall$ i : 0 $\leq$ i $\wedge$ i < #a : a.i = ar.s.i)

with ar.s.n defined for 0 <= n < #a by

```
ar.s.n =        letting s = x ~ X
                if (n = #x) -> X
                 (n < #x) -> ar.x.(n-1)
                fi
```

**Theorem 1:** The following transformation converts an array into an equivalent sequence:

var a: array of T

var s: sequence of T with #s = #a

; s = seq.a.#a

with seq.a.n defined for $0 \leq n \wedge n < \#a$ by

seq.a.n = if (n = 0) $\rightarrow \varepsilon$

(n > 0) $\rightarrow$ seq.a.(n-1) ~ [a.(n-1)]

fi

**Proof:** To prove the equivalence, we show that $(\forall\ i : 0 \leq i < \#a : a.i = ar.(seq.a.\#a).i)$

We use the following definition of a sub-array.

**Definition 3:** b is a sub-array of a if for a, b: array of T with #b < #a the following holds

$(\forall i : 0 \leq i \wedge i < \#b : b.i = a.i)$

It is easily seen that ar.(seq.a.#a).(#b-1) = ar.(seq.b.#b).(#b-1) since the definition of seq.a.n does not involve array elements, a.i with i > n-1. We can, therefore, work with a sub-array with #b = i+1

ar.(seq.a.#a).i

= < choosing to look at a sub-array with #b = i+1>

ar.(seq.a.#a).(#b-1)

= < using the above observation>

ar.(seq.b.#b).(#b-1)

= < using the definition of seq.a.n with n > 0 since i ≥ 0 hence #b > 0 >

ar.(seq.b.(#b-1) ~ [b.(#b-1)]).(#b-1)

= < using the definition of ar with n = #x since #b-1 = #b-1 >

b.(#b-1)

= < using the sub-array definition >

a.(#b-1)

= < using i = #b-1>

a.i

## A.5. Bit-vector Representation

We will now look at improving the tree-based representations. A natural modification consists of representing the child-node purity within the parent node. The parent-node purity is then redundant except for the root node that has no parent to maintain its purity information.

**Definition 4:** A P-vector-tree is a tuple, tv

(fv, lv, pv, zv, chv),

where pv and zv are an array of bool, with #pv = #zv = f. A P-vector tree can be defined based on a P-tree by the transformation v: t $\rightarrow$ tv with

fv = f

lv = l-1

($\forall$ i : 0 $\leq$ i $\wedge$ i < #ch : pv.i = ch.i.p)

($\forall$ i : 0 $\leq$ i $\wedge$ i < #ch : zv.i = ch.i.z)

chv array of P-vector-tree with ch.i $\rightarrow$ chv.i using tranformation v

This transformation preserves all information of a P-tree except for the root node. We, therefore, introduce the concept of a root parent.

**Definition 5:** The root parent is a tuple (f, l, p, z, ctv) where f denotes the fan-out, l stands for the level, p is a Boolean variable that is true if the tree is a pure 1 tree, z is a Boolean variable that is true if the tree is a pure 0 tree, and ctv is an P-vector tree with lv = l-1 and fv = f.

In implementations, the situation can be resolved by allowing a pure root node to contain child vectors of length f with their corresponding purity information. The performance benefit of using a P-vector-tree lies in the possibility of interpreting pv and zv as bit-vectors that can internally be represented by integers or arrays thereof. We will use the term child-purity vector or bit-vector with the understanding that they are mathematically represented by arrays of type bool. Storage can be reduced by the following observations. Pure nodes no longer hold information that is not represented by their parent: by definition, f and l can be derived from the parent, and the child vector of a pure node by definition had length 0. Therefore, #pv = #zv = 0.

**Definition 6:** A redundant tree, rt, is a P-vector tree with #pv = #zv = 0.

Redundant trees do not have to be represented. Since level l=0 in definition 1 is entirely redundant, it can be removed; therefore, lv = l-1. Consequently, the level with lv = 0 in pv does not have children and $(\forall\ i : 0 \leq i \wedge i < f : pv.i = \neg zv.i)$. Note that the transformation to the P-vector representation thereby reduces the total number of levels by 1, with only the single root parent being added. Eliminating redundant trees corresponds to the following transformation of the child array. Instead of the array ch, we use a sequence, cs, that is defined as

cs = dens.ch.0

dens.ch.n = if (n = f) → ε

(n < f) → if (ch.n.#pv = 0) → dens.ch.(n+1)

(ch.n.#pv ≠ 0) → [ch.n] ~ dens.ch.(n+1)

fi

fi

Using the array-sequence equivalence introduced earlier, we may now replace child array chv with a compressed child array, cca, to represent child node information, where we define cca to be equivalent to the child sequence, cs. It is very important for the efficiency of computations that the compressing array, cca, can efficiently be mapped to the array non-compressing array ch:

$(\forall\, i : (0 \le i \wedge i < f\,) \wedge (ch.i.\#pv \neq 0)$: $ch.i = cca.(\underline{N}\, j : 0 \le j \wedge j < i : (\neg pv.j \wedge \neg zv.j)))$ ,

where N is the count operator defined in [1]. Compare Chapter 3 for the efficient evaluation of counts.

## A.6. Dense P-trees

We saw that representing child information within the parent node has significant benefits both for the computational and the storage complexity. We can continue this strategy to further reduce the need for pointers in implementations. Each child-purity vector can itself be represented in its parent, resulting in the representation of grandchild purity within each node. Each parent holds an array of the child-purity vectors of its children. The array can either be compressing or non-compressing as in the case of the child arrays in the P-vector representation: pure nodes do not have children; therefore, if a child is pure, grandchild purity does not have to be recorded. As was the case when moving to child-purity vectors, the new purity vectors, the grandchild purity, make the previous ones, the child purity, redundant except for the root node.

**Definition 7:** A dense P-tree is a tuple, td

(fd, ld, pd, zd, chd),

where pd and zd are arrays of array of bool, with ($\forall$ i : $0 \le i \wedge i < \#pd$ : #pd.i = f) and ($\forall$ i : $0 \le i \wedge i < \#zd$ : #zd.i = f). A P-vector tree can be defined based on a P-vector tree by the transformation d: tv $\rightarrow$ td with

fd = f

ld = lv-1

($\forall$ i : $0 \le i \wedge i < \#chd$ : pd.i = chd.i.p)

($\forall$ i : $0 \le i \wedge i < \#chd$ : zd.i = chd.i.z)

chd array of dense P-tree with chv.i $\rightarrow$ chd.i using using transformation d

This transformation preserves all information of a P-vector tree except in the case of the root node. We, therefore, adapt the concept of a root parent to hold the entire ancestry. Note that the formulation of the root ancestry is very unpleasant. It is not, however, possible to omit the root ancestry from dense implementations entirely. In practice, it may be possible to treat part of the root ancestry information as a dense node with #chd = 1. This does not, however, make the mathematical description any more transparent and will, therefore, be omitted.

**Definition 8:** The root ancestry is a tuple (f, l, p, z, pv, zv, ctd), where f is of type integer and denotes the fan-out and l is of type integer and stands for the level. The level is chosen such that, in the case of pure ancestry, the number of binary digits in the sequence is $f^l$, matching the definition of the original P-tree. p is a Boolean variable that is true if the tree is a pure 1 tree; z is a Boolean variable that is true if the tree is a pure 0 tree; pv is the child purity vector with #pv = f and #zv = f for ($\neg p \wedge \neg z$); #pv = 0 and #zv = 0 for ($p \vee z$); and ctd is a dense P-tree with lv = l-1, fd=f.

We can again consider nodes with #pd = #zd = 0 redundant and eliminate them. The lowest level is completely eliminated by that reasoning. The argument proceeds in full analogy to P-vector trees and will not be repeated.

## A.7. Array-converted Dense P-trees

Dense P-trees have many useful properties. Easy access to children is combined with very reasonable storage requirements and efficient computations based on bit vectors. One problem that remains is that the natural implementation of dense P-trees uses pointers to represent child nodes. Pointers have undesirable properties, especially in an environment that uses distributed processing. We will, therefore, now look at a way to represent P-trees entirely through arrays. Based on definition 7 of a dense P-tree, we construct a pre-order sequence

pre.td = if (#chd = 0) → [td]

(#chd > 0) → [td] ~ chdpre.td.0

fi

chdpre.td.n= if (n = #chd-1) → pre.(td.ch.(#chd-1))

(n < #chd-1) → pre.(td.ch.n) ~ chdpre.(td.(n+1))

fi

The array, ada, is defined to be equivalent to pre.td. We can now augment td with ai: array of int, #ai = #chd to get the following modified tree definition, tdmod:

(fd, ld, pd, zd, chd, ai),

where $(\forall\ i : 0 \leq i \wedge i < \#chd : (ai.i = j) \wedge (ada.j = chd.i))$, resulting in tda. ai.i holds the address of the $i^{th}$ child of the node that contains ai, which allows us to move through the

array in the same way as through a tree without using trees as part of trees that would commonly be represented by pointers.

The final definition of an array-converted dense P-tree holds the array index information as part of the information at each address. Based on the type node (ai: array of int, pd: array of array of bool, and zd: array of array of bool), the ad tree can be written as ad = pre.tda with

pre.tda = if (#chd = 0) → [(ai, pd, zd)]

(#chd > 0) → [(ai, pd, zd)] ~ chdpre.td.0

fi

chdpre.tda.n= if (n = #chd-1) → pre.(td.ch.(#chd-1))

(n < #chd-1) → pre.(td.ch.n) ~ chdpre.(td.(n+1))

fi

## A.8. Comparison with Implemented Code

The theoretical description naturally required some adaptation from the implemented Java data structures and algorithm implementations. Note that Figure 3.7 shows an example tree of the implementation. The following list gives a summary of the most important differences:

- None of the code is recursive as the recurrence relations may suggest.
- The code does not represent pure 0 information (z) but rather mixed information (m=¬(p∨z)).
- Count information is maintained at each node.
- Parallel arrays are used for pure 1, mixed, address, and counts rather than one array with tuples as elements.

- Array elements are child-purity vectors with all child-purity vectors that relate to one node stored in sequence, which leads to a slight modification in the evaluation of array indices. The mathematical formulation uses two numbers to identify node elements, the array index and the position within the node. The Java implementation uses a single index that follows naturally from the order in which node elements are listed.
- The root ancestry is modeled as a node with #chd = 1.
- Root ancestry that corresponds to an entirely pure tree leads to a representation using f pure child trees to avoid the exceptional treatment using a single bit.

The main concepts are, however, accurately described.

## A.9. References

[1] Edward Cohen, "Programming in the 1990s: An Introduction to the Calculation of Programs," Springer-Verlag, New York, 1990.