Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Pavel Šanda

Implicit propositional proofs

Department of Applied Mathematics

Supervisor: Prof. RNDr. Jan Krajíček, DrSc.
Study Program: Computer Science

Prague 2006

Acknowledgements

First of all, I would like to thank my supervisor Jan Krajíček for careful guidance, patience and support.
Also, I would like to thank my parents, for encouraging me through the whole studies.

I declare that I have written this master thesis on my own and listed all used sources. I agree with lending of the thesis.

Prague, 23rd March 2006

Pavel Šanda

Thesis was defended on 22nd May 2006.

Název práce: Implicitní výrokové důkazy
Autor: Pavel Šanda
Katedra: Katedra aplikované matematiky
Vedoucí diplomové práce: Prof. RNDr. Jan Krajíček, DrSc., MÚ AV

Abstrakt: Hlavním cílem této práce je navrhnout algoritmus, který bit po bitu popíše důkazy konkrétní množiny tautologií v daném důkazovém systému. Budeme studovat především tautologie, které formalizují Dirichletův princip. Pro tyto tautologie lze sestrojit krátké důkazy v obvyklých Fregeovských systémech, ale neexistuje možnost podat takovýto krátký důkaz v rezolučním důkazovém systému. Existuje však možnost popsat tento dlouhý rezoluční důkaz "implicitně", pomocí polynomiálního algoritmu. Naše konstrukce umožňuje sestrojit důkaz korektnosti algoritmu, aniž bychom dopředu předpokládali platnost Dirichletova principu. Toho lze využít pro formulaci silnějšího důkazového systému, nežli je rezoluce samotná. Jak bude vysvětleno později, existence takovýchto systémů s krátkými důkazy úzce souvisí s podstatnými otázkami v teorii složitosti a studium silných důkazových systémů může pomoci porozumění těmto otázkám.

Klíčová slova: implicitní výrokový důkaz, Dirichletův princip.

Title: Implicit propositional proofs
Author: Pavel Šanda
Department: Department of Applied Mathematics
Supervisor: Prof. RNDr. Jan Krajíček, DrSc., MÚ AV

Abstract: The main goal of this thesis is to propose an algorithm which bit by bit describes proofs of a particular set of tautologies in a certain proof system for propositional logic. In particular, we will study tautologies formalizing the pigeonhole principle (PHP). These tautologies have short proofs in usual Frege proof systems, but there is no short proof in the resolution proof system. However, there is a possibility how to “implicitly” describe a long resolution proof by a polynomial-time algorithm. In addition, our construction allows to prove the soundness of the algorithm without proving PHP first. This can be used for formulation of stronger proof system than resolution itself. As it will be described in the following chapters, the existence of proof systems with short proofs is tightly connected with fundamental questions in complexity theory and the study of strong proof systems can help us in understanding these questions.

Keywords: implicit propositional proof, pigeonhole principle.

Contents

CHAPTER 1

Introduction

There are essentially two views of the concept of a mathematical proof. The usual one is a mixture of formal and natural language and the only criterion for its correctness is the agreement between mathematicians (represented by the referee process in mathematical journals). The second one consists of the precise formalisation of the logical language and manipulating operations upon it, established by the work of Frege and other past logicians. In this formalization we can consider a mathematical theorem as a string of symbols and the task of proving it becomes exactly defined string-manipulations. However, such a formalization can be done in many different ways and one basic task of proof theory is the classification of these distinct approaches. The formalization usually consists of the underlying logical language formulation, the axioms-formulation and certain inference rules, which constitute the possible consequences of the axioms - i.e. the proof strength of the selected proof system. The next task is to study the structure and the size of the formal proofs in such a system, which is mainly the task of proof complexity theory.

This paper is devoted to the construction of proofs of certain tautologies in one particular proof system recently suggested in Krajíček[**1**]. The structure of the thesis is the following: in the second chapter we introduce the concept of a proof system due to Cook-Reckhow[**2**], and discuss its relation to computational complexity. In the same chapter we will define, in particular, the resolution proof system, which will be used for proving pigeonhole principle. The proof itself and the algorithm describing it will be formulated in chapter 3. In chapter 4 we will prove soundness and discuss how the previous proofs relate to implicit proofs.

1.1. Notation

$[n]$ denotes the set $\{1, \ldots, n\}$.

$\sum^*$ denotes the set of words over a non-empty alphabet $\sum$; this includes also the empty word.

$|w|$ denotes the length of a string w .

CHAPTER 2

Proof systems for propositional logic

2.1. Propositional logic

2.1.1. Syntax.

Definition 2.1.1. **Language** L of propositional logic consists of :

- Propositional variables, which represent the atomic propositions (*atoms*) with the two possible values - TRUE and FALSE. Atoms are usually denoted by p_i .
- Symbols for some k-ary boolean functions

  $$f : \{TRUE, FALSE\}^k \to \{TRUE, FALSE\}\,,$$

  which are usually called connectives. In this paper we will use the following symbols: $\vee$ (disjunction), $\wedge$ (conjunction), $\overline{}$ (negation) and constants 0/1, which are 0-ary functions with FALSE/TRUE values.
- Auxiliary symbols: (,) .

Propositional formulas are build inductively using rules:

- Any propositional variable or constant is formula.
- If A, B are formulas then $\overline{A}$, $(A \vee B)$, $(A \wedge B)$ are also formulas.

Any propositional formula is built using the above rules finitely many times.

2.1.2. Semantics.

Definition 2.1.2. **Truth assignment** for set V of propositional variables is a mapping $\alpha : V \to \{TRUE, FALSE\}$. Such an assignment can be extended to all formulas of the language L built from atoms in V using the usual definition for connectives.

A formula is a **tautology** iff its evaluation gives TRUE for any possible assignment to the atoms. We shall denote the fact that A is a tautology by $\models A$.

For T a set of formulas we denote by $T \models A$ the fact that every truth assignment which satisfies T (i.e. all formulas in T) satisfies also A.

Let $TAUT$ denote the set of all tautologies in L.

2.2. Propositional proof systems

Definition 2.2.1. (Cook-Reckhow[**2**]) Let us fix some alphabet $\sum$, where $|\sum| \geq 2$. **A propositional proof system** (*pps*) is any polynomial-time computable function $P : \sum^* \rightarrow L^*$, where $Rng(P) = TAUT$.

For tautology $A \in TAUT$, any string $w \in \sum^*$ such that $P(w) = A$ is called a **P-proof** of A .

A *pps* P is **polynomially bounded** if there exists a polynomial $p(x)$, such that any tautology A has a P-proof w of size $|w| \leq p(|A|)$.

Definition 2.2.2. (Cook-Reckhow[**2**]) Let $F_1 : \sum_1^* \rightarrow L^*$ and $F_2 : \sum_2^* \rightarrow L^*$ be pps. Then F_2 **p-simulates** F_1 iff there is polynomial time computable function $g : \sum_1^* \rightarrow \sum_2^*$, such that $F_2(g(w)) = F_1(w)$ for all $w \in \sum_1^*$.

The connection between computational complexity and proof theory is described by the following two theorems.

Theorem 2.2.3. (Cook[**3**]) $P = NP \Leftrightarrow TAUT \in P$.

Thus, for example, in the case of $P = NP$, we could construct polynomially bounded proof system, for which the given tautology A represents also the proof of A.

The second theorem is closely related to the efficiency of proof systems. It shows that the definition of the poly-bounded pps is another way of characterizing NP class, which explains some of the motivations of proof system study.

Theorem 2.2.4. (Cook-Reckhow[**2**]) $NP = coNP \Leftrightarrow$*there exists a polynomially bounded propositional proof system (* $\Leftrightarrow TAUT \in NP$*).*

Proof. If $NP = coNP$ then $TAUT \in NP$. Then there exists function P which for a given certificate C confirm tautology A in time polynomial in $|A|$, so w.l.o.g. $|C|$ is polynomial to $|A|$. In other words certificate (C, A) is a proof in the requested polynomially bounded proof system P .

Conversely let us have polynomially bounded proof system P . Then the proofs of this system are certificates in an $NP - definition$ of $TAUT$. If $TAUT \in NP$ holds, $coNP = NP$ follows, as $TAUT$ is a $coNP - complete$ problem. $\square$

2.3. Frege proof systems

In the case of Frege proof systems, the formalization of inference rules and axioms falls under one common definition:

Definition 2.3.1. A **Frege rule** is a system of formulas $A_0, \dots, A_k$ written as:

$$\frac{A_0, \dots, A_{k-1}}{A_k}$$

where $A_0, \dots, A_{k-1} \models A_k$.

A Frege rule in which k=0 is called a **Frege axiom scheme**.

An instance of the rule is obtained by a simultaneous substitution of arbitrary formulas for the atoms in $A_0, \dots, A_k$.

An **Inference system** F is a finite set of Frege rules.

Definition 2.3.2. (Cook-Reckhow[**2**]) If F is inference system then:

An **F-proof** of formula B from formulas $A_1, \dots, A_n$ is a finite sequence $C_1, \dots, C_k$ of formulas, such that $C_k = B$, and such that C_i is either one of $A_1 \dots, A_n$ or is inferred from some earlier C_j's ($j < i$) by a rule of F . We denote it as $A_1, \dots, A_n \vdash_F B$ or simply $A_1, \dots, A_n \vdash B$, if F is clear from the context.

F is **implicationally complete** iff

$A_1, \dots, A_n \vdash_F B$ whenever $A_1, \dots, A_n \models B$.

F is a **Frege proof system** iff F is implicationally complete.

Note that the implicational completeness of the proof system is often deduced from its particular properties, namely:

- **Soundness** : $\vdash A \Rightarrow \models A$
- **Completeness** : $\models A \Rightarrow \vdash A$

The most often used Frege proof systems are those based on modus ponens inference rule and the DeMorgan language. As an example we will show a familiar Frege system.

Example 2.3.3. $\mathcal{F}$ proof system. Language has only two connectives $L_{\mathcal{F}} = \{\rightarrow, \bar{}\}$. All other DeMorgan connectives can be defined in $L_{\mathcal{F}}$. The only inference rule could be schematically written as:

$$\frac{A, A \rightarrow B}{B}$$

Axiom schemes $A_{\mathcal{F}}$:

$$A \rightarrow (B \rightarrow A)$$

$$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$$

$$(\overline{B} \rightarrow \overline{A}) \rightarrow (A \rightarrow B)$$

THEOREM 2.3.4. *$\mathcal{F}$ from the Example 2.3.3 is a proof system.*

PROOF. We will construct a function P in the sense of the Definition 2.2.1. For an input string w , the function P has to :

(1) Check the syntax form of w , i.e. whether $w = C_1, \ldots, C_k$. As there are unambiguous rules for the construction of formulas, we can find and check every C_i by a simple decompositional algorithm in a time polynomial to $|w|$.
(2) Check the correctness of the derivation of each C_i . For this P firstly test, whether given C_i is an instance of an axiom scheme from $A_{\mathcal{F}}$. This can be done in time polynomial to $|C_i|$. If C_i is not such an instance, then P will test whether C_i was created by the inference rule. For this P will try to infer C_i from some pair C_j, C_m , $m, j < i$. As there is only $O(k^2)$ pairs and each test of inference rule can be done in time polynomial to $|C_i| + |C_j| + |C_m|$, the whole inference-test can be done in time polynomial to $|w|$, because $i, |C_i|, |C_j|, |C_m| \leq |w|$. If both tests fail, C_i is not constructed via Frege rules defined in $\mathcal{F}$ and the semantic check fails.
(3) When both checks are satisfied P will return C_k as an output (i.e. C_k is tautology and w is its proof), otherwise P will return constant 1 (i.e. w is not proof in the sense of the Definition 2.3.2 and P returns tautology 1 to keep $Rng(P) = TAUT$). Obviously, this task can be performed also in time polynomial to $|w|$.

As three previous computations are all polynomial to $|w|$, the whole function P is computable in time polynomial to $|w|$.

Now we show that $Rng(P) = TAUT$.

(1) By the truth-table method we can easily check, that any formula constructed from $A_{\mathcal{F}}$ axiom schemes is tautology. Furthermore, the inference rule is sound: any assignment which satisfies the hypotheses $A, A \rightarrow B$ satisfies also the consequence B. Now, because P accepts only proofs which correctly use axioms and inference rule, any output formula from P is tautology, i.e. $Rng(P) \subseteq TAUT$.
(2) To prove, that $TAUT \subseteq Rng(P)$ we need, for a given tautology A, to find a particular string (i.e. proof), which will accepted by P . This follows from the completeness of $\mathcal{F}$, proof of which can be found in any introductory book for propositional logic.

□

Note, that the first part is just a formulation of the soundness and the second one of the completeness of $\mathcal{F}$. Thus, we can see, that the condition $Rng(P) = TAUT$ in the Definition 2.2.1 is an abstract formulation of soundness and completeness in the Definition 2.3.2.

THEOREM 2.3.5. (Reckhow[**4**]) *Assume that F and F' are two Frege systems and that their languages are complete (i.e. all DeMorgan connectives can be expressed there).*

Then F and F' polynomially simulates each other.

As a consequence of this theorem it follows that one Frege system over a complete language is polynomially bounded iff all Frege systems over complete languages are.

In the effort to increase the strength of the proof system additional rules can be added as in the following definition - however, whether the following system is really stronger is still an open question and some results as Theorem 3.2.3 show the possibility, that their strength is equivalent.

DEFINITION 2.3.6. (Cook-Reckhow[**2**])Let us have some Frege system F.

An **Extended Frege proof** of formula B is a finite sequence of formulas $C_1, \ldots, C_k$, such that $C_k = B$, and such that C_i is either inferred from some earlier C_j's by an F-rule of F or has the form of an **extension axiom:**

$$q \equiv D$$

where the **extension atom** q is not previously used (including formula D) in the proof and it is not used in C_k .

In other words, a proof constructed in this way allows to abbreviate formulas inside the proof and this way reduce its length. However, we can still check such a proof in polynomial time, hence it forms a proof system, which we denote as an **Extended Frege proof** system eF.

2.4. Resolution

In the rest of the thesis we confine ourselves to the resolution proof system, usually denoted R. It works only on the constrained set of formulas, which are in the DNF form, thereby we can base the whole system on a very simple language. The constrain on the DNF form is not fatal and other formulas can be transformed in DNF by the means of the so-called limited extension. The next property of resolution is that it tries to refute the negation of the given proposition instead of directly proving it. Thus proving certain formula in resolution usually means to take its negation and transform it to the CNF form. For this reason R is sometimes called a refutation system.

2.4.1. Language. A **literal** is an atom (propositional variable) or its negation. A **clause** is a disjunction of literals, which is usually written in the set-notation $\{l_1, \ldots, l_r\}$. A clause can be empty. A clause is satisfied by a truth assignment if at least one literal in it is. In particular, the empty clause is unsatisfiable.

A set of initial clauses represents the CNF formula, which is to be refuted. The empty clause represents the desired contradiction, as there is no satisfying assignment to it.

2.4.2. Inference rules. The **resolution rule** is :

$$\frac{C \cup \{p_i\} \qquad D \cup \{\overline{p_i}\}}{C \cup D}$$

The atom p_i is called the **resolved atom** and the conclusion is called the **resolvent** of $C \cup \{p_i\}$ and $D \cup \{\overline{p_i}\}$. Without loss of generality we can assume that neither p_i nor $\overline{p_i}$ occur in $C \cup D$.

It is easy to see, that the rule is sound: if the hypotheses of resolution inference are simultaneously satisfiable, then also the resolvent is satisfiable by the same assignment. As the empty clause is not satisfiable, it lead us to the definition of resolution refutation:

Definition 2.4.1. (Blake[**5**]) Let us have a formula in a DNF form $A = \bigvee_{i=1}^{n} B_i$, where $B_i = \bigwedge_{j=1}^{n_i} l_{i,j}$ and $l_{i,j}$ are literals. Now for each subformula B_i form clause $C_i = \left\{\overline{l_{i,j}} | j \leq n_i\right\}$. A sequence of clauses $D_1, \ldots, D_k$ is then a resolution proof of A (or **resolution refutation** of $C_1, \ldots, C_n$) iff

- each D_t is either an initial clause C_i or clause inferred from some D_l, D_m, $l, m < t$ by the resolution rule.
- D_k is the empty clause.

2.4.3. Soundness and completeness.

Theorem 2.4.2. *Let A be a DNF formula. Then* $\models A \Leftrightarrow \vdash_R A$.

Proof. Soundness: Let us have the refutation of initial clauses constructed from A. As the resolution inference rule is sound, any resolution refutation preserves satisfiability of the initial clauses. That implies, that the end clause is also satisfied, which is impossible for any assignment, as there is nothing to satisfy in the empty clause. It follows that there is no assignment satisfying all initial clauses. Hence A is a tautology.
Completeness: Let us have the set of unsatisfiable clauses $\mathcal{C} = \{C_1, \ldots, C_n\}$ constructed from some tautology in the way described in the definition above. We will prove by induction on the number of r variables appearing in $\mathcal{C}$ (i.e. $\mathcal{C}$ consists of $l_1, \ldots, l_r, \overline{l_1}, \ldots, \overline{l_r}$ literals), that any such $\mathcal{C}$ has a

refutation. The case of $r = 0$ is trivial. Assume $r > 0$. We will find a set of clauses $\mathcal{C}'$ such that:

(1) each clause in $\mathcal{C}'$ is derivable from $\mathcal{C}$
(2) $\mathcal{C}'$ is unsatisfiable
(3) At most $(l-1)$ different variables appear in $\mathcal{C}'$

Let x be some variable in $\mathcal{C}$, then we construct:

$$\mathcal{C}_x = \{C_i | x \in C_i \wedge \overline{x} \notin C_i\}$$
$$\mathcal{C}_{\overline{x}} = \{C_i | \overline{x} \in C_i \wedge x \notin C_i\}$$
$$\mathcal{C}_\emptyset = \{C_i | x \notin C_i \wedge \overline{x} \notin C_i\}$$
$$\mathcal{C}_\cup = \left\{ D \,\middle|\, \frac{C_x \qquad\qquad C_{\overline{x}}}{(C_x \backslash \{x\}) \cup (C_{\overline{x}} \backslash \{\overline{x}\}) \;\; (\equiv D)} \;,\; C_x \in \mathcal{C}_x, C_{\overline{x}} \in \mathcal{C}_{\overline{x}} \right\}$$

$$\mathcal{C}' = \mathcal{C}_\emptyset \cup \mathcal{C}_\cup$$

Firstly $\mathcal{C}'$ does not contain variable x. That implies, that we have transformed our case l into the case $l-1$, where the induction hypothesis holds.

Secondly, $\mathcal{C}'$ is unsatisfiable: suppose there is a truth assignment which satisfies $\mathcal{C}'$. This implies that either all clauses $C_x \backslash \{x\}$ or all $C_{\overline{x}} \backslash \{\overline{x}\}$ are satisfied, otherwise we can find a combination from both sets, which could not be satisfied in $\mathcal{C}_\cup$. W.l.o.g. assume all $(C_x \backslash \{x\})$ are satisfied. We extend the assignment by setting $x := 0$, so now the whole $\mathcal{C}$ is satisfied. But that contradicts the assumption that $\mathcal{C}$ is unsatisfiable.

It follows, by induction hypothesis, that $\mathcal{C}'$ has a refutation and as our transformation used only resolution rule, $\mathcal{C}$ has also a refutation. □

2.5. The strength of proof systems

To sum up, we show in the following figure a relation between proof systems described earlier. The arrow from A to B indicates that A can polynomially simulate B.

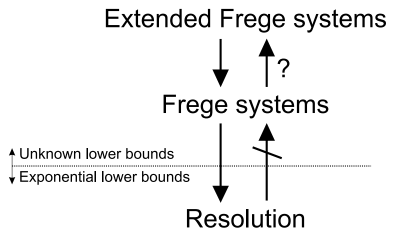


FIGURE 2.5.1. Hierarchy of the proof systems

CHAPTER 3

The pigeonhole principle

3.1. Definition

The pigeon-hole principle is a very simple combinatorial statement. Its informal form states that having $n-1$ holes and n pigeons sitting in the holes, there is some hole with more than one pigeon. The usual formalization in combinatorics is that there is no one-to-one mapping from a set of n elements into a set of $n-1$ elements.

In the context of propositional proofs, the formalization encodes this mapping (or relation) into propositional formulas using boolean variables p_{ij}, where $p_{ij} = 1 \Leftrightarrow$ *pigeon i is in hole j* . The universal quantifier is transformed into the set of the formulas connected by conjunctions while the existential quantifier is represented by disjunctions.

Since we will work in the resolution proof system, we will take the negation of this principle and encode individual parts as can be seen in the following table:

Part	Relation		Clause
1.	totality	for every pigeon there exists some hole	For every i, $1 \leq i \leq n$, include clause $\bigvee_{1 \leq j \leq n-1} p_{ij}$
2.	injectivity	for each hole: for every pair of two pigeons, one of those pigeons is not placed in the hole	For every i, j, k, $1 \leq j \leq n-1$, $1 \leq i < k \leq n$ include clause $(\neg p_{ij} \vee \neg p_{kj})$
3.	surjectivity	for every hole there exists some pigeon	For every j, $1 \leq j \leq n-1$, include clause $\bigvee_{1 \leq i \leq n} p_{ij}$
4.	function	for every pigeon: for every pair of two holes, one of those holes does not contain the pigeon	For every i, j, k, $1 \leq i \leq n$, $1 \leq j < k \leq n-1$ include clause $(\overline{p_{ij}} \vee \overline{p_{ik}})$

TABLE 1. PHP parts

Various conjunctions of the constrains from the previous table lead to the propositional definition of various pigeonhole principles :

$$\neg PHP^n_{n-1} \equiv 1 \wedge 2$$

$$\neg onto - PHP^n_{n-1} \equiv 1 \wedge 2 \wedge 3$$

$$\neg functional - PHP^n_{n-1} \equiv 1 \wedge 2 \wedge 4$$

$$\neg symmetric - PHP^n_{n-1} = 1 \wedge 2 \wedge 3 \wedge 4$$

Up to now we consider $n-1$ holes and $m=n$ pigeons. Another variety of the pigeonhole principles can be obtained for $m \gg n$ pigeons. Generally these cases (denoted as $weak - PHP$) are easier to prove and many complexity results are known, but we will not study these variants here.

3.2. Proofs

3.2.1. Overview. First we recall a few results concerning the proof complexity of PHP .

THEOREM 3.2.1. (Cook-Reckhow[**2**]) *The upper bound for a proof of PHP^n_{n-1} in the Extended Frege system is $n^{O(1)}$.*

THEOREM 3.2.2. (Haken[**6**]) *The lower bound for a proof of symmetric$-$ PHP^n_{n-1} in the resolution proof system is $exp(\Omega(n))$.*

THEOREM 3.2.3. (Buss[**7**]) *The upper bound for a proof of PHP^n_{n-1} in the Frege proof system is $n^{O(1)}$.*

By the Theorem 3.2.2 PHP requires exponential size proofs in resolution and indeed, the proof we construct has exponential size too.

However, our aim in the following chapters is to describe this proof via a polynomial-time algorithm, so the proof system, which uses this algorithm as part of its proof can be more powerful than the resolution itself.

One obvious question is, why we introduce new exponential size proof of the PHP tautologies, when we have constructively built the resolution refutation for any tautology in the completeness part of the Theorem 2.4.2. The point is that the soundness of the construction in the proof of Theorem 2.4.2 (i.e. that it constructs a resolution proof) is derived from the fact that the formula is a tautology. The proof of the soundness of our construction does not need to assume this.

REMARK. From now on, we will talk about initial clauses of PHP, meaning really of $\neg PHP$, which is quite customary in the area. However there is no danger of confusion.

3.2.2. Metaproof. We will construct the proof of PHP^n_{n-1} by induction on n. In the induction step we remove the last pigeon and the last hole, i.e. from

$$f : \{1, \dots, n\} \to \{1, \dots, n-1\}$$

we restrict

$$f' : \{1, \dots, n-1\} \to \{1, \dots, n-2\} \ ,$$

where

$$f'(i) = \begin{cases} f(i) \ \ when\ f(i) \neq n-1 \\ \quad f(n) \ \ otherwise \end{cases}$$

Thus if f is one-to-one then f' is also one-to-one. As each step removes one pigeon and one hole, the whole induction proceed to the case of two pigeons sitting in one hole, which has the proof of the constant length - in the language of the resolution : $\{p_{1,1}\}, \{p_{2,1}\}, \{\overline{p_{1,1}}, \overline{p_{2,1}}\}, \{\overline{p_{2,1}}\}, \{\}$.

3.2.3. Resolution proof. Since we will manipulate a lot with the axioms (initial clauses) during the proof, we introduce an additional notation.

DEFINITION 3.2.4. Let $M \subseteq [n]$ denote a set of pigeons and $N \subseteq [n-1]$ denote a set of holes. We encode the pigeonhole principle for pigeons from M and holes from N via two sets of clauses as suggested in Section 3.1, namely

$$PHP_1(M, N) \equiv \{\{p_{i,j} \mid j \in N\} \mid i \in M\}$$

$$PHP_2(M, N) \equiv \{\{\overline{p_{i,k}}; \overline{p_{j,k}}\} \mid i, j \in M; i \neq j; k \in N\}$$

Then define

$$PHP(M, N) \equiv PHP_1(M, N) \cup PHP_2(M, N) \ \ .$$

We define the special case $PHP_1([1], \emptyset) \equiv \{\emptyset\}$.

Sometimes we will use just PHP_1 or PHP_2 , when the sets M and N will be obvious from the context.

So the PHP^n_{n-1} from the previous paragraphs becomes $PHP([n],[n-1])$. As will be described below, the part PHP_1 will be transformed as the proof will proceed, while the clauses in PHP_2 part remain untouched. So the PHP_2 part will be tacitly included in $PHP(M,N)$ and often not discussed at all.

Next we need to index somehow the k-th axiom in the PHP_1 part, hence we introduce the following clause:

Definition 3.2.5. $PHP(M,N)[k] \equiv c$, such that

$$c \in PHP_1(M,N) \ \&\ \exists p_{k,j} \in c\,,$$

for some arbitrary j. In particular $PHP([1],\emptyset)[1] \equiv \{\emptyset\}$.

In the case of $PHP([n],[n-1])[k]$ we get $\{p_{k,j} \mid j \in [n-1]\}$.

The Idea of the resolution proof. We would like to prove PHP^n_{n-1} by induction. In such a case the basic task is to convert PHP^i_{i-1} into PHP^{i-1}_{i-2} and use that PHP^{i-1}_{i-2} can be refuted by the induction hypothesis.

More precisely, one wants to derive the clauses of PHP^{i-1}_{i-2} from the clauses PHP^i_{i-1} and then complete the refutation by appending a refutation of PHP^{i-1}_{i-2} , which exists by the induction hypothesis. As pointed out, the PHP_2 clauses of PHP^{i-1}_{i-2} are a subset of the PHP_2 clauses of PHP^i_{i-1} , so nothing needs to be done to deduce them.

Although the idea is not difficult, the formalization is technically more intricated, so for the demonstration purposes we firstly show the transformation step from n to $n-1$. Later on we formulate it precisely and generally for the i-th induction step.

Let us have $PHP([n],[n-1])$. We will derive the empty clause from the last axiom $PHP([n],[n-1])[n] \equiv \{p_{n,1},\dots,p_{n,n-1}\}$ via sequential resolving with literals $\overline{p_{n,x}}$, $x \in [n-1]$ (i.e. with clauses $\{\overline{p_{n,x}}\}$). Now we focus on the question how to obtain these literals $\overline{p_{n,x}}$. Firstly we introduce the $\oplus$ operator:

Definition 3.2.6. Let $g_1,\dots,g_j$ be some literals. Then define

$$PHP(M,N) \oplus g_1 \oplus \dots \oplus g_j$$

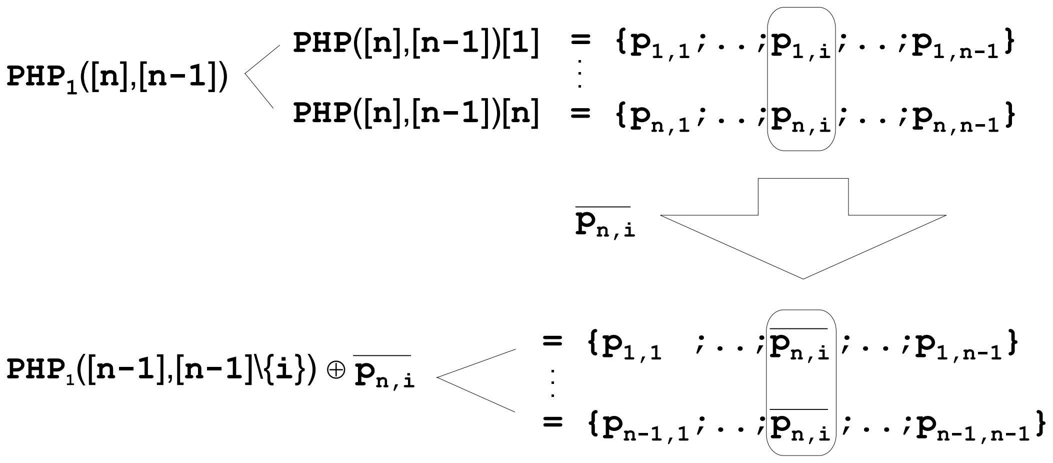
as

$$\{x \cup \{g_1,\dots,g_j\} | x \in PHP_1(M,N)\} \cup PHP_2(M,N)\,.$$

We call any of the g_i's a **side literal**.

Similarly we denote

$$PHP(M,N)[k] \oplus g_1 \oplus \dots \oplus g_j \equiv \{x \cup \{g_1,\dots,g_j\} | x \in PHP(M,N)[k]\}\ .$$



$PHP_1([n],[n-1])$
$PHP([n],[n-1])[1] = \{p_{1,1}; \ldots; p_{1,i}; \ldots; p_{1,n-1}\}$
$PHP([n],[n-1])[n] = \{p_{n,1}; \ldots; p_{n,i}; \ldots; p_{n,n-1}\}$
$\overline{p_{n,i}}$
$PHP_1([n-1],[n-1]\backslash\{i\}) \oplus \overline{p_{n,i}}$
$= \{p_{1,1}; \ldots; \overline{p_{n,i}}; \ldots; p_{1,n-1}\}$
$= \{p_{n-1,1}; \ldots; \overline{p_{n,i}}; \ldots; p_{n-1,n-1}\}$

FIGURE 3.2.1. Transformation of PHP_1 clauses

Further, we say that $PHP(M,N) \oplus g_1 \oplus \cdots \oplus g_j$ is **side-refutable** iff $\{g_1, \ldots, g_j\}$ can be derived from it (i.e. $PHP(M,N)$ is refutable). If $P \oplus g$ is side-refutable, we denote g as **refutant**.

Now consider the following operation (see Figure 3.2.1): each clause $PHP([n],[n-1])[k]$ ($\in PHP_1$) resolve with the clause $\{\overline{p_{k,x}}, \overline{p_{n,x}}\}$ ($\in PHP_2$) . The resulted set of the derived clauses can be in our notation described as $PHP_1([n-1],[n-1]\backslash\{x\}) \oplus \overline{p_{n,x}}$. The clauses of $PHP_2([n],[n-1])$ remain the same , but only their subset $PHP_2([n-1],[n-1]\backslash\{x\})$ will be chosen . If we ignore the side literal, then in case of $x = n-1$ we get precisely $PHP([n-1],[n-2])$ and for the others $x's$ we get the same instance of PHP only with the renamed indexes.

So, if we are able to refute $PHP([n-1],[n-2])$, we automatically get the resolution steps for deriving the side literal, because to derive the empty clause from $PHP([n-1],[n-1]\backslash\{x\})$ yields automatically a derivation of $\overline{p_{n,x}}$ from $PHP([n-1],[n-1]\backslash\{x\}) \oplus \overline{p_{n,x}}$ - the side literal is untouched by the resolution rule and "bubles up" through the whole refutation process. Thus using the side-refutability of $PHP([n-1],[n-1]\backslash\{x\}) \oplus \overline{p_{n,x}}$ we sequentially get all negative literals needed for the refutation of $PHP([n],[n-1])$.

The general case of $PHP_1(M,N)$ differs from our demonstration only in the fact that we get a (possibly) renamed set of holes and some side literals, which have been added in the previous steps of resolution, but the operation remains the same.

Our procedure leads us to the tree-structure (as can be seen in the Figure 3.2.2) of the whole proof, where each vertex is equivalent to some set $PHP([n-i],[n-1]\setminus\{v_1,v_{2,}\dots,v_i\})\oplus\overline{p_{n,v_1}}\oplus\overline{p_{n-1,v_2}}\cdots\oplus\overline{p_{n-i+1,v_i}}$; each edge corresponds to the resolved literal $\overline{p_{n-j+1,v_j}}$, which is to be found as the side literal in the vertices below and $v_1,\dots,v_i$ actually describes a path in the tree. The *final* clause $PHP([1],[n-1]\setminus\{v_1,\dots,v_{n-1}\})\oplus\overline{p_{n,v_1}}\oplus\cdots\oplus\overline{p_{2,v_{n-1}}} = \{\overline{p_{n,v_1}},\dots,\overline{p_{2,v_{n-1}}}\}$ is always a leaf vertex of the tree.

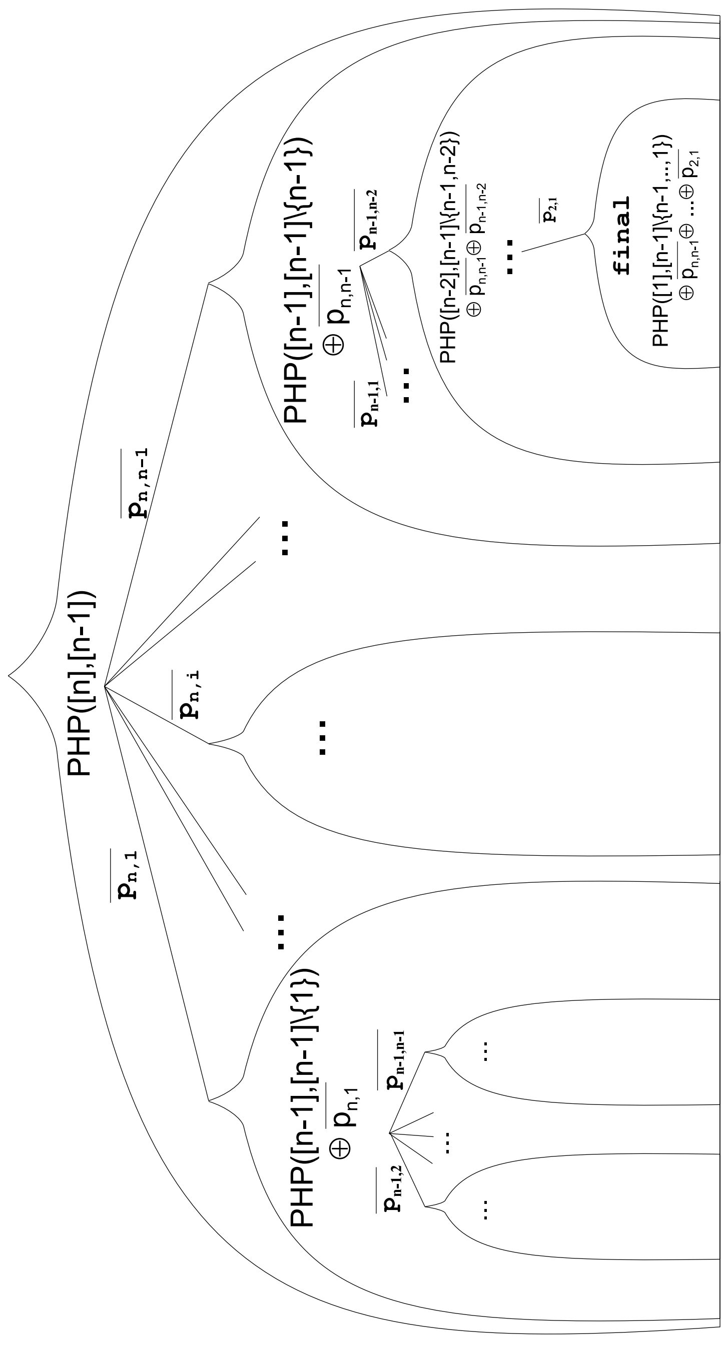

$PHP([n],[n-1])$
$\overline{p_{n,1}}$
$\overline{p_{n,i}}$
$\overline{p_{n,n-1}}$
$PHP([n-1],[n-1]\backslash\{1\}) \oplus \overline{p_{n,1}}$
$\overline{p_{n-1,2}}$
$\overline{p_{n-1,n-1}}$
$PHP([n-1],[n-1]\backslash\{n-1\}) \oplus \overline{p_{n,n-1}}$
$\overline{p_{n-1,1}}$
$\overline{p_{n-1,n-2}}$
$PHP([n-2],[n-1]\backslash\{n-1,n-2\}) \oplus \overline{p_{n,n-1}} \oplus \overline{p_{n-1,n-2}}$
$\overline{p_{2,1}}$
final
$PHP([1],[n-1]\backslash\{n-1,...,1\}) \oplus \overline{p_{n,n-1}} \oplus ... \oplus \overline{p_{2,1}}$

FIGURE 3.2.2. Structure of the proof

3.3. P-tree & DFS

Now we will describe our tree formally and make its relation to the eventual resolution proof explicit (see fig. 3.3.1).

The tree (denote it *p-tree*) is not a proof itself, but only an auxiliary structure, which helps to describe the actual resolution proof. After we describe the *p-tree*, we can conceive two algorithms:

(1) First one is the classical DFS[1] algorithm (denote it *alg-dfs*), which has the *p-tree* as an input and produces resolution steps as an output. When it steps into the vertex v from the vertex u through edge e it dumps out the resolution steps transforming v into the u according the information associated with e. In this way the algorithm gradually prints the whole resolution proof and the correctness of the resolution proof can be easily seen, because DFS give us ordering of the vertices of the *p-tree*. Further, we fix the ascending order of edges (according to the associated values) outcoming from vertex u in which will *alg-dfs* evaluate u.

(2) However the algorithm used in the implicit proof (denote it *alg-ip*) has the index of a place (w.l.o.g. a literal or a whole clause) in the resolution proof as an input and the literal (resp. the clause) at the place as an output. Thus the output of *alg-ip* for all the indexes gives the same as *alg-dfs*. Moreover, the algorithm has to be fast enough, i.e. it has to allow to generate a resolution step without a computation of the previous resolution steps. The details will be given in section 3.5.

3.3.1. *p-tree* description. The proof tree has depth n. The root of the tree (1-st level) consists of axioms $PHP([n],[n-1])$.
Each vertex on level d is associated (consists of) with set of $n-d+1$ clauses $PHP_1([n-d+1],[n-1]\backslash\{v_1,\ldots,v_{d-1}\})$, which has been resolved from the parent vertex. To simplify the description we denote such a vertex on level d as *t-vertex*, where $t = n-d+1$.
From each t-vertex originate *t-1* edges (not counting the edge from the parent), each one being associated with one of $\overline{p_{t,z}}$ ($z \in [n-1] \setminus \{v_1,\ldots,v_{d-1}\}$).
Each edge from t-vertex to (t-*1*)-vertex (child) with its associated literal $\overline{p_{t,z}}$ will substitute $p_{i,z}$ in each (t-*1*)-vertex clause (i.e. *z-th* "column" in t-vertex clauses will be replaced by column filled with $\overline{p_{t,z}}$, see Figure 3.2.1). Leafs of the tree are the vertices on the n-th level, i.e. there is only one clause associated with the vertex and consists only of the literals associated with the edges on the path to the leaf.

[1]depth-first-search

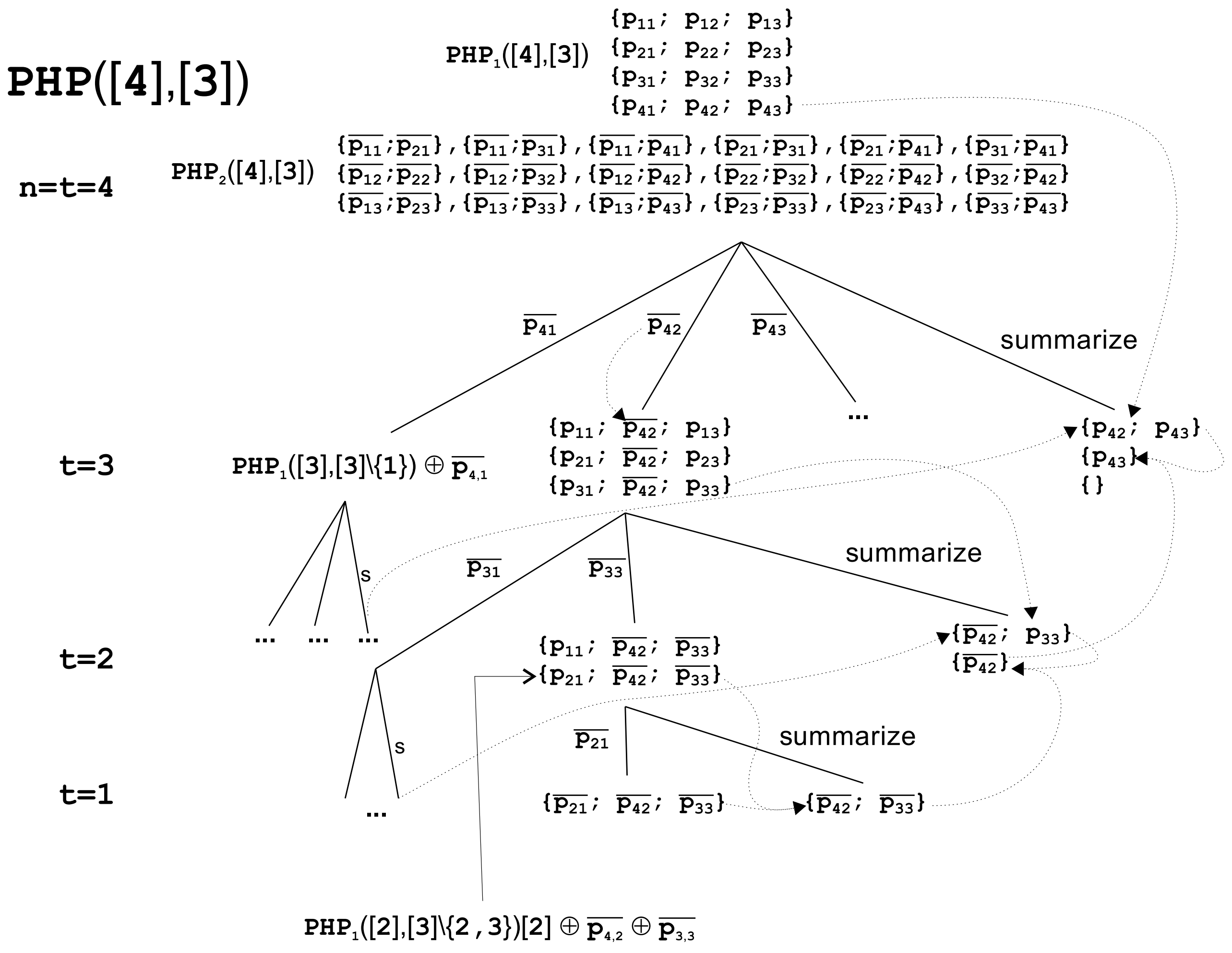


FIGURE 3.3.1. p-tree

Till now, we could conceive the whole resolution proof as two phases procedure:

(1) As *alg-dfs* output on our tree, where the step-downwards from the (root) vertex into some lower-level vertex via edge e means that we are resolving all clauses in that vertex with the literal associated with edge e, while the step-upwards does nothing.
(2) In this way we 'prepared' all the leaves in the tree, and now we have to derive the empty clause from them.

However, it is more appropriate to join these two steps together, so all the resolution steps relating to the last axiom of the current vertex will be located on the same place. This can be done by changing the upwards-step of the *alg-dfs*. Each upward-step on the edge associated with $\overline{p_{t,z}}$ will be associated with resolution steps, which are used to remove $\overline{p_{t,z}}$ from the leaf-clause (or its smaller consequence which gradually arise as we move upwards to the root). These resolution steps consist of resolving the last axiom clause $PHP_1([t],[n-1]\backslash\{v_1,\dots,v_{d-1}\})[t]$ of t-vertex gradually with *t-1* leaf-clause-derivates returned by the upward-step from child (*t-1*)-vertices. So finally at the root we obtain the empty clause.

To simplify the precise description of the proof we introduce a new kind of *summary* vertex in the *p-tree* as can be seen in Figure 3.3.2. For each t-vertex, besides *t-1* child vertices we add one summary vertex. The unique edge of the summary vertex will be associated with the 'upward' resolution steps described above. When we associate these steps with the edge from the parent-vertex to summary(child)-vertex, we can still apply DFS, which produce resolution steps, by down-stepping into the vertex; in this way we also obtain a linear ordering of resolution steps associated with the edges in the tree.

3.3.2. Addressing. Now we define a concept of an address in the tree.

Definition 3.3.1. The **address** in the p-tree is a two-component vector $\vec{a_v} = (\vec{x},\vec{y})$, where $\vec{x} = (v_1,\dots,v_k,0,\dots,0)$ determine the path from the root to vertex v via $\overline{p_{n,v_1}},\dots,\overline{p_{n-k+1,v_k}} - edges$. In case of the path to the summary vertex, we introduce the symbol s (i.e. $v_k = s$).
Vector $\vec{y} = (i,j)$ determine, the j-th atom in the i-th clause of vertex v $(PHP([n-k],[n-1]\backslash\{v_1,\dots,v_k\})[i])$.

We define the **index** of the address to be

$$\|\vec{a_v}\| = zn^3 + \sum_{i=1}^{2} y_{3-i} * z^i + \sum_{i=1}^{n} x_{n-i+1} * z^{i+2} \ ,$$

where $z = 2 + max(n, max\ length\ of\ clause\ in\ p-tree)$.

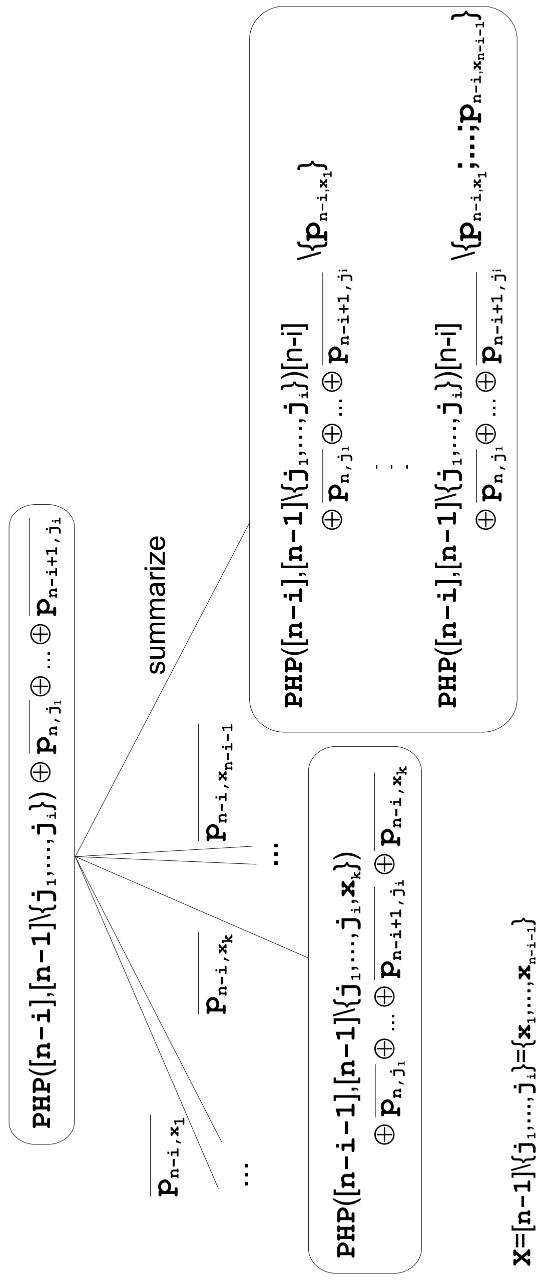

PHP([n-i],[n-1]\{j_1,…,j_i}) ⊕ p_{n,j_1} ⊕ … ⊕ p_{n-i+1,j_i}
summarize
PHP([n-i],[n-1]\{j_1,…,j_i})[n-i] ⊕ p_{n,j_1} ⊕ … ⊕ p_{n-i+1,j_i} \{p_{n-i,x_1}}
PHP([n-i],[n-1]\{j_1,…,j_i})[n-i] ⊕ p_{n,j_1} ⊕ … ⊕ p_{n-i+1,j_i} \{p_{n-i,x_1},…,p_{n-i,x_{n-i-1}}}
p_{n-i,x_1}
p_{n-i,x_k}
p_{n-i,x_{n-i-1}}
PHP([n-i-1],[n-1]\{j_1,…,j_i,x_k}) ⊕ p_{n,j_1} ⊕ … ⊕ p_{n-i+1,j_i} ⊕ p_{n-i,x_k}
X=[n-1]\{j_1,…,j_i}={x_1,…,x_{n-i-1}}

FIGURE 3.3.2. Structure of the vertices in p-tree. The first $n-i-1$ branches from parent vertex serve to obtain negative literals, which will be used in the summarizing branch to reduce their positive counterpart in last $(n-i)$-*th* clause of the parent vertex.

For symbol s we reserve the value of $z-1$. The maximal length of the PHP clause can be estimated by $10n\lceil log\, n\rceil$.

The final resolution proof will have on the indexes corresponding to an address $\vec{a}_v$ the clause v, while the rest of the proof will be padded with some auxiliary symbol (padding literal). The range $1,\dots,zn^3$ of the index values is reserved for indexing the PHP_2 clauses.

We define the ordering of the vertices by $v<u \Leftrightarrow \|\vec{a_v}\| < \|\vec{a_u}\|$.

Observation: When *alg-dfs* reaches vertex v before u, then $u<v$.

Corollary. *If* alg-dfs *dumps resolution steps correctly (in the sense, that each dumped clauses can be resolved from the previous dumped clauses), addressing will index atoms in the final resolution proof and the resolutions steps in that proof will be also correct.*

3.4. Detailed description of the p-tree

In this section we give a precise description of the vertices in the *p-tree*. We picked a notation, which uses only very simple constructions, in such a way that from it can be straightforwardly constructed an algorithm we shall need later. It can be also clearly seen, that for constructing some inner vertex of the p-tree, we do not need any additional computation which would evaluate the previous resolution steps. This is a crucial demand on our algorithm, as we need to find an effective algorithm describing the full resolution proof.

Notation 3.4.1. Here we describe the conventions and the notation used in the Table 2. Common symbols are:
Used "columns" $U:=\{j_1,\dots,j_{k-1}\}$. All "columns" $R:=[n-1]$.
$P_z^X:=\{p_{z,i}|i\in X\}$. P_i^R then actually means $PHP_1([n],[n-1])[i]$.
We will address $\{\overline{p_{a,b}};\overline{p_{c,b}}\}$ simply be denoting $PHP_2(a,b,c)$ instead of coding it into the first zn^3 values by some arbitrary formula.

We will classify all the vertices in the p-tree. Each classified family consists of associated resolution rules and description of each resolution rule will have the following structure:

```
Clause 1
Clause 2
⊢ Clause 3
Addresses
Address 1
Address 2
⊢ Address 3
```

with the obvious meaning, that Clause 1, Clause 2 on addresses Address 1, Address 2 can be resolved into Clause 3 with Address 3, which is actually the address of the currently classified vertex.

In order to write the formulas generally for all cases of vertices we make a rule, that whenever we should denote a non-existing literal (e.g. $\{p_{i,0}\}$) , it will be ignored (i.e. $\emptyset$) - thus $(j_1, \dots, j_{k-1}, z, \dots)$ for $k = 1$ denotes $(z, \dots)$.

There are two kinds of numbers which appear in the address as can be seen in the Figure 3.4.1. While a plain number (e.g. j_i) in the address denotes the edge associated with the corresponding literal-number, the number of the form $\wr i \wr$ denotes the *i-th* edge arising from the vertex. These two kinds of denotations can be easily transformed into each other in time polynomial to n. When $\wr i \wr$ appears in literal, it has the same value, as it would have after the index j_{k-1} in the address of currently investigated vertex class.
The symbol s in the address denotes the edge to the summary vertex, $\wr n - k + 1 \wr$ in other words.

As we count the pigeons from 1 to n we can use number 0 in the address for the recognition of the end of the path in the p-tree (i.e. $j_i \neq 0$). We will write the termination as "$0, \dots$" because the following numbers are not important as far as the p-tree is concerned. A bolded or an underlined font is used just for a better comprehension and has no syntactical consequence.

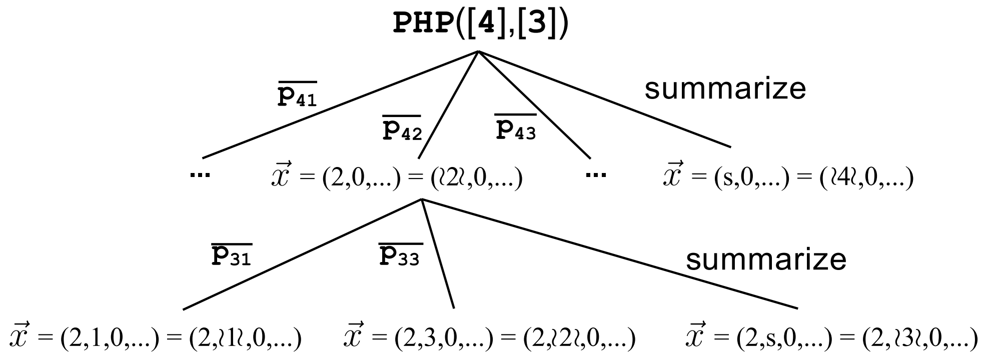

PHP([4],[3])
$\overline{p_{41}}$
$\overline{p_{42}}$
$\overline{p_{43}}$
summarize
...
$\vec{x}$ = (2,0,...) = ($\wr$2$\wr$,0,...)
...
$\vec{x}$ = (s,0,...) = ($\wr$4$\wr$,0,...)
$\overline{p_{31}}$
$\overline{p_{33}}$
summarize
$\vec{x}$ = (2,1,0,...) = (2,$\wr$1$\wr$,0,...)
$\vec{x}$ = (2,3,0,...) = (2,$\wr$2$\wr$,0,...)
$\vec{x}$ = (2,s,0,...) = (2,$\wr$3$\wr$,0,...)

FIGURE 3.4.1. Correspondence between $\wr i \wr$ and j_k in address

- standard vertex
  1. *st-v1:* standard vertex with $\vec{x} = (j_1, \dots, j_k, 0, \dots)$
     Associated resolution rules
     $\forall z \in [n-k] : \{\overline{p_{n-k+1,j_k}}; \overline{p_{z,j_k}}\}$,
     $\left(P_z^R \backslash P_z^U\right) \cup \{\overline{p_{n-i+1,j_i}} \mid i \in [k-1]\}$
     $\vdash \left(P_z^R \setminus \left(P_z^U \cup \{p_{z,j_k}\}\right)\right) \cup \{\overline{p_{n-i+1,j_i}} \mid i \in [k]\}$
     Addresses
     $PHP_2(n-k+1, j_k, z)$,
     $(\vec{x} = (j_1, \dots, j_{k-1}, \underline{0}, 0, \dots), \vec{y} = (z, \dots))$
     $\vdash (\vec{x} = (j_1, j_2, \dots, j_k, 0, \dots), \vec{y} = (z, \dots))$

- summary vertex
  1. *sum-v1*: inner summary with $\vec{x} = (j_1, \dots, j_{k-1}, j_k \equiv s, 0, \dots)$
     First part of associated resolution rules
     $\left(P_{n-k+1}^R \setminus P_{n-k+1}^U\right) \cup \{\overline{p_{n-i+1,j_i}} \mid i \in [k-1]\}$,
     $\{\overline{p_{n-i+1,j_i}} \mid i \in [k-1]\} \cup \{\overline{p_{n-k+1,\wr 1 \wr}}\}$
     $\vdash \left(\left(P_{n-k+1}^R \setminus P_{n-k+1}^U\right) \cup \{\overline{p_{n-i+1,j_i}} \mid i \in [k-1]\}\right) \setminus \{p_{n-k+1,\wr 1 \wr}\}$
     *Address*es
     $(\vec{x} = (j_1, \dots, j_{k-1}, 0, \dots), \vec{y} = (n-k+1, \dots))$,
     $(\vec{x} = (j_1, \dots, j_{k-1}, \wr 1 \wr, s, 0, \dots), \vec{y} = (n-k-1, \dots))$
     $\vdash (\vec{x} = (j_1, \dots, j_k, 0, \dots), \vec{y} = (1, \dots))$
     sum-v2: Second part of associated resolution rules
     $\forall z \in [n-k-1]$
     $\left(\left(P_{n-k+1}^R \setminus P_{n-k+1}^U\right) \cup \{\overline{p_{n-i+1,j_i}} \mid i \in [k-1]\}\right) \setminus \{p_{n-k+1,\wr t \wr} | t \in [z]\}$,
     $\{\overline{p_{n-i+1,j_i}} \mid i \in [k-1]\} \cup \{\overline{p_{n-k+1,\wr z+1 \wr}}\}$
     $\vdash \left(\left(P_{n-k+1}^R \setminus P_{n-k+1}^U\right) \cup \{\overline{p_{n-i+1,j_i}} \mid i \in [k-1]\}\right) \setminus \{p_{n-k+1,\wr t \wr} | t \in [z+1]\}$
     *Address*es
     $(\vec{x} = (j_1, \dots, j_k, 0, \dots), \vec{y} = (z, \dots))$,
     $(\vec{x} = (j_1, \dots, j_{k-1}, \wr z+1 \wr, s, 0, \dots), \vec{y} = (n-k-1, \dots))$
     $\vdash (\vec{x} = (j_1, \dots, j_k, 0, \dots), \vec{y} = (z+1, \dots))$
  2. *sum-v3*: leaf summarizing vertex with $\vec{x} = (j_1, \dots, j_{n-2}, \wr 2 \wr)$
     Has the same formulas. However, $k = n-1$ and thus second iterative part through z will not be present. The second address in the first part is $(\vec{x} = (j_1, \dots, j_{k-1} \equiv j_{n-2}, \wr 1 \wr, 0, \dots), \vec{y} = (1))$.
  3. *sum-v4*: root summarizing vertex with $\vec{x} = (j_1 \equiv s, 0, \dots)$
     Has the same formula. Addresses of $(\vec{x} = (\underline{0}, \dots), \vec{y} = (i))$ we define as axiom $PHP([n], [n-1])[i]$.
     The desired empty clause has address $(\vec{x} = (s, 0, \dots), \vec{y} = (n-1, \dots))$.

TABLE 2. **Classification of the vertices**

3.5. Alg-ip

Up to now we have showed how the address from the *p-tree* will be converted into the index in the final resolution proof. In fact, *alg-ip* algorithm has to do just the opposite - from a given index it will decode a proper address and further from the decoded address it will compute the proper clause with desired literal; this is possible according to the formulas given in the Table 2.

However, the output of the algorithm will be only a subset of the information given in the Table 2 - in particular, for given Address 3, the output will have the form of "Address 1,Address 2,Clause 3". Adding the antecedents addresses for each clause in the resulted resolution proof is in no conflict with the definition of resolution, moreover it simplifies the proof of correctness.

Both addressing and *alg-ip* were formulated for the case of the one-symbol-output. However, when considering the Table 2 and the proof of correctness we work with the whole clauses, which is more convenient and do not bring any essential change.

Note also, that much space of the resolution proof will be filled with the padding literal, which will be output of *alg-ip* in case that the input number cannot be converted into the correct address (i.e. pointer to existing clause) of the p-tree. Although there would be some loss as far as the length of the proof is concerned, we gain the possibility of an efficient computation of the clause when given its address.

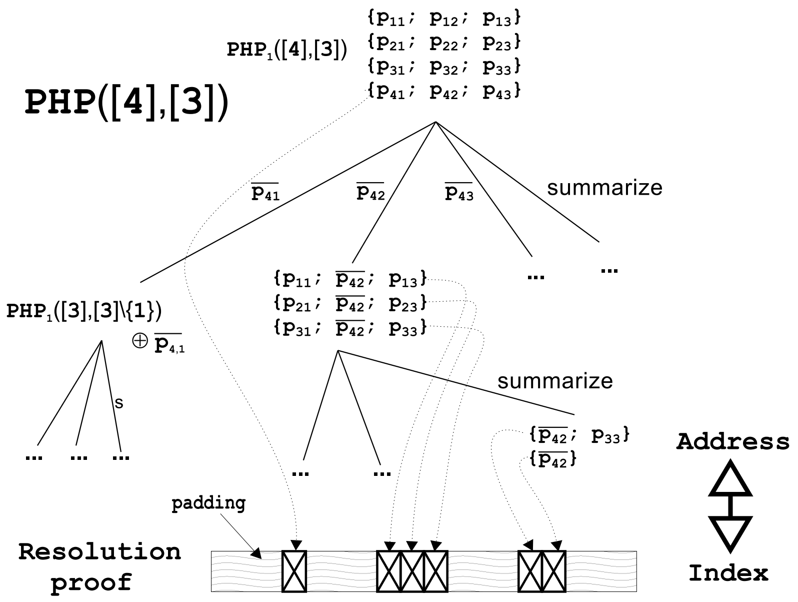


FIGURE 3.5.1. Address - to - index transformation

3.5.1. Outline of *alg-ip*. Here we give a skeleton, how would be the algorithm *alg-ip* constructed. Note that this sketch is for demonstration purpose only and the implementation should be done more efficiently.

```
Index_to_Clause(index $a$, size $n$)
  if $(a \leq zn^3) \Rightarrow PHP_2$
  else
    address := Index_to_Address($a$,$n$);
    clause := Generate_vertex(address,$n$);
    result := clause[$a.y_2$];
end

Index_to_Address(index $a$,size $n$)
  $a := a - zn^3$;
  $y_1 := a \, mod \, z$;  $a := a/z$;
  $y_2 := a \, mod \, z$;  $a := a/z$;
  for ($i = 1$; $a > 0$ ; $i = i + 1$)
   $x_i := x \, mod \, z$;
   $a := a/z$;
  end
  result := $(\vec{x}, \vec{y})$;
end

 //Generate the vertex according to the Table 2
Generate_vertex(address $a$,size $n$)
 if ($a \sim summary \, vertex$)
           result := generate_summary_clause($a$,$n$);
  else
   if ($a \sim ordinary \, vertex$)
            result := generate_ordinary_clause($a$,$n$);
    else
       result := padding_literal;
end
```

CHAPTER 4

Soundness and complexity demands

4.1. Relation to implicit proofs

In this chapter we will prove the soundness of the refutation algorithm. We will also measure the complexity demands. That is important for two reasons - firstly we will check the correctness of the *alg-dfs* given by the Table 2, secondly we will give by this proof informal description of the correctness proof used as a part of an implicit proof.

The idea introduced in Krajíček[**1**] is that we construct a proof system, which instead of using exponentially long proof for a given tautology supplies a circuit of polynomial size, which outputs the required proof bit-by-bit. To fulfil the conditions in the Definition **2.2.1** of a proof system (in particular we need that it can be recognized in p-time whether a string is a proof or not), we have to check somehow the correctness (optimally in the time polynomial to the size of the given circuit). This cannot be done simply by checking the output of the algorithm for being a correct proof, because such evaluation will lead to the exponential time complexity (here we take the circuit size as the input). We can avoid this difficulty, when we prove the correctness of the algorithm's computation - then if the input and the last output formula is correct, we know without the evaluation that the whole proof generated by the algorithm is correct. Hence we take the pair of algorithm and its correctness proof as proof itself and denote a proof system based on such pair-proofs as *implicit proof system.*

In our case there would be the possibility (which we do not pursue here formally) to construct the implicit proof system based on the proof-pairs of *alg-ip* and correctness proof from the Section 4.3 written in some formal calculus. From the Section 4.2 we can see, that such a proof system would be polynomially bounded on the class of PHP-tautologies and thus more strong then the resolution proof system.

4.2. Complexity demands

4.2.1. General estimates. Our language contains constant number of symbols used for constructions of clauses. A number in an address or a literal in a clause used for description of pigeons and holes in $PHP([n],[n-1])$ has space complexity of $O(log(n))$ - because the logarithmic factor will not affect (non)polynomiality of computations, we will count its complexity as $O(1)$. Also when we speak about polynomial or exponential time we mean it with respect to n. In the estimates below we distinguish particular polynomial classes, which is not necessary as we need to distinguish only polynomial and exponential complexity classes.

There is $O(n^2)$ "initial" literals used for initial clauses of PHP. All literals in the inferred clauses during the proof originate from these initial literals, so one clause has the length at most $O(n^2)$. In case of the clauses in *p-tree* we get even the $O(n)(= O(z))$ estimate.

An address has the length of $O(n)$, thus a complete output *"a_1, a_2, c_3"* for a given index from *alg-ip* gives $O(n)$ length.

The number of clauses in the resolution proof can be estimated from the size of p-tree - its depth, number of descendants of each vertex and number of clauses in each vertex are all at most n, so we get exponential number of clauses in the proof. $\vec{a_{last}} = (\vec{x} = (s \equiv n, 0, \dots), \vec{y} = (n-1, \dots))$ is the address of the last clause, and $\|\vec{a_{last}}\|$ gives us the highest index used for input of *alg-ip* - $O(z^n)$, z comes from Definition 3.3.1. Thus we can see, that the input of *alg-ip* has at most $O(n)$ symbols.

4.2.2. alg-ip. Now we will consider *alg-ip* algorithm, because a polynomial time complexity of *alg-ip* implies a polynomial size of the circuit computing *alg-ip*. Firstly we have to convert an index into the address format. This can be done straightforwardly in $O(n)$ time. After that we have to recognize the class of a given vertex, which can be done according to the Table 2 in a polynomial time. Construction of one clause in any vertex family consists of the following operations:

- Evaluating P^X_y sets. This can be done in $O(n)$.
- Basic set operations on evaluated sets, which can be done in $O(n^2)$.
- Conversion of index-type number (e.g. $\wr x \wr$), which can be done in $O(n^2)$ time.

All of the above operations are used only constant-times, so we can conclude that the time complexity of the whole *alg-ip* is polynomial.

4.3. Soundness

Notation 4.3.1. We will use the Notation 3.4.1 from the Table 2. Moreover, for a given vertex with the address $a.\vec{x} = (j_1, \dots, j_k, 0, \dots)$ we will sometimes use the index j_{k+1} (or higher) with the meaning of the first (or higher) unused number in the address - in other words: with given address $a.\vec{x}$, j_{k+h} corresponds to the value $\wr h \wr$ computed from the address $a.\vec{x} = (j_1, \dots, j_k, \wr h \wr, 0, \dots)$.

THEOREM 4.3.2. *alg-ip is correct.*

Firstly we introduce a notion of the proof tree. The output[1] of *alg-ip* for all indexes (i.e. previously shown resolution refutation of PHP) can be presented as a proof tree. Each vertex represents some output clause from *alg-ip* and (two) incoming edges from vertices V_1, V_2 to some vertex V_3 represent one resolution inference $V_1, V_2 \vdash_R V_3$. Thus leafs of the proof tree are initial clauses for refutation and the root of the tree is the empty clause.

To prove the soundness of *alg-ip* we need to prove that it produces a correct proof in the sense of Definition 2.3.2, in other words to check these three properties:

(1) The root vertex of the proof tree is the empty clause (formally speaking: $alg - ip(\|a_{last}\|).c_3 = "\{\}")$.
(2) The leafs are the initial clauses of PHP.
(3) We can resolve each vertex (except the leafs) from some two previous vertices. Moreover, we can compute these previous vertices (and their positions) explicitly knowing only the position of the current one and in a polynomial time.

All the vertices in the proof tree originate from the clauses of p-tree vertices. Thus if we check correctness of the rules described in Table 2, which defines the p-tree, we verify correctness of the proof tree also, because the property "V_3 is correctly inferred from V_1, V_2" is preserved in the translation from the p-tree to the proof-tree.

PROOF. Firstly we fix the convention used in the proof. Usually the sub-proof of a single case of a vertex with address a_3 will have the following structure:

$$\begin{aligned}
&vertex - class, a_3 \equiv address\\
&a_1 \equiv address_1\\
&a_2 \equiv address_2\\
&\vdash c_3 \equiv clause_3\\
&alg - ip(a_1).resolvent \equiv c_1\\
&alg - ip(a_2).resolvent \equiv c_2
\end{aligned}$$

For a given vertex with the address a_3 we simply compute a_1, a_2, c_3 from the Table 2 - which is nothing but the output of $alg - ip(a_3)$. Now for proving the correctness, we have to compute clauses c_1, c_2 from a_1, a_2, resolve them by resolution rule and compare, whether the resolvent is identical with c_3, in other words to check, whether the rules in Table 2 are correct. Furthermore,

[1]We will not consider the padding symbols in output of *alg-ip*, as they have no meaning for the correctness of the proof.

this is done not using particular numerical values of indexes but rather several properties of addresses - otherwise it will lead to the exponential number of checks.

Now we will check the three previously mentioned properties.

1. The root of the proof tree corresponds to the last clause of the last-visited summary vertex of the p-tree, which has the following address: $(\vec{x} = (s \equiv n, 0, \dots), \vec{y} = (n-1, \dots))$. By Table 2 we can express the resultant clause $c_3 \equiv \left(P_n^R \backslash P_n^{\emptyset} \cup \emptyset\right) \setminus \{p_{n,\wr t \wr} | t \in [n-1]\} = \{\}$.

2. All the vertices in the proof tree are translated from the p-tree, and as the translation preserves antecedents in the resolution rules, we can determine the leafs of proof tree with the help of Table 2. Analyzing the table we can see that all the clauses in the p-tree has antecedents, except:

(1) Clauses in a vertex with address $\vec{x} = (0, \dots)$.

(2) Clauses denoted as $PHP_2(x, y, z)$.

Now, because $\vec{x} = (0, \dots)$ is address of PHP_1, all the clauses without antecedents are clauses of PHP, thus all the leafs of the proof tree are initial clauses.

3. To check the inner vertices of the proof tree, we check for the correctness systematically all the classes of generic vertices in the p-tree. As mentioned above we use here only the general properties of the addresses, not any particular numerical values.

- *st-v1*, $a_3.\vec{x} = (j_1, \dots, j_k, 0, \dots)$.

$\forall z \in [n-k]$:
$a_1 \equiv PHP_2(n-k+1, j_k, z)$
$a_2 \equiv (\vec{x} = (j_1, \dots, j_{k-1}, 0, \dots), \vec{y} = (z, \dots))$
$\vdash c_3 \equiv \left(P_z^R \backslash \left(P_z^{\{j_1,\dots,j_{k-1}\}} \cup \{p_{z,j_k}\}\right)\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [k]\}$
$= \left\{p_{z,j_{k+1}}; \dots; p_{z,j_{n-1}}; \overline{p_{n,j_1}}; \dots; \overline{p_{n-k+1,j_k}}\right\}$
$c_1 = \{\overline{p_{n-k+1,j_k}}; \overline{\mathbf{p_{z,j_k}}}\}$
$c_2 = \left(P_z^R \backslash \left(P_z^{\{j_1,\dots,j_{k-2}\}} \cup \{p_{z,j_{k-1}}\}\right)\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [k-1]\}$
$= \left\{\mathbf{p_{z,j_k}}; \dots; p_{z,j_{n-1}}; \overline{p_{n,j_1}}; \dots; \overline{p_{n-k,j_{k-1}}}\right\}$

As we can see c_2 has additional p_{z,j_k} and does not have $\overline{p_{n-k+1,j_k}}$ compared to c_3, so c_1 is exactly needed for a correct resolution inference $c_1, c_2 \vdash_R c_3$.

- *sum-v1,* $a_3 = \left(\vec{x} = (j_1, \dots, j_{k-1}, j_k \equiv s, 0, \dots), \vec{y} = (1, \dots)\right)$.

$$\begin{aligned}
&a_1 \equiv \left(\vec{x} = (j_1, \dots, j_{k-1}, 0, \dots), \vec{y} = (n-k+1, \dots)\right)\\
&a_2 \equiv \left(\vec{x} = (j_1, \dots, j_{k-1}, \wr 1 \wr, s, 0, \dots), \vec{y} = (n-k-1, \dots)\right)\\
&\vdash c_3 \equiv \left(\left(P^R_{n-k+1} \backslash P^{\{j_1,\dots,j_{k-1}\}}_{n-k+1}\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [k-1]\}\right) \backslash \{p_{n-k+1,\wr 1 \wr}\}\\
&= \left\{p_{n-k+1,j_{k+1}}; \dots; p_{n-k+1,j_{n-1}}; \overline{p_{n,j_1}}; \dots; \overline{p_{n-k+2,j_{k-1}}}\right\}\\
&c_1 = \left(P^R_{n-k+1} \backslash \left(P^{\{j_1,\dots,j_{k-2}\}}_{n-k+1} \cup \left\{p_{n-k+1,j_{k-1}}\right\}\right)\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [k-1]\}\\
&= \left\{\mathbf{p_{n-k+1,j_k}}; \dots; p_{n-k+1,j_{n-1}}; \overline{p_{n,j_1}}; \dots; \overline{p_{n-k+2,j_{k-1}}}\right\}
\end{aligned}$$

Note, that a_1 is the address of the last clause of a standard-type vertex from a higher level in the p-tree. This last clause is the only clause from $a_1.\vec{x}$ vertex, which has not been used up to now in contrast to the rest of the clauses in $a_1.\vec{x}$. Thus we do not break the tree-condition of proof tree - any clause is still used as an antecedent only once.

Evaluating c_2 is bit more complicated. Because a_2 is an address of a summary-type clause we should distinguish two cases:

- whether $a_2.\vec{y}[1] = 1$(*sum-v1 type*)
- or $a_2.\vec{y}[1] > 1$ (*sum-v2 type*).

However, these two cases differ only in the antecedent pointer of c_2, not in the way how the resolvent clause c_2 is expressed. As we only need c_2 clause to check $c_1, c_2 \vdash_R c_3$, we can check both cases in one common expression of c_2.
To avoid a confusion we distinguish between indexes $(j_1, \dots, j_{k-1}, \wr 1 \wr, s)$ used when describing the vertex with address a_3 and identical indexes $(j'_1, \dots, j'_k, s)$ used when evaluating c_2. Thus $p_{n-k,\wr t \wr}$ is here computed as if $\wr t \wr$ appear in address $(j'_1, \dots, j'_k, \wr t \wr)$.

$$\begin{aligned}
c_2 &= \left(\left(P^R_{n-k} \backslash P^{\left\{j'_1,\dots,j'_{k-1},j'_k \equiv \wr 1 \wr\right\}}_{n-k}\right) \cup \left\{\overline{p_{n-i+1,j'_i}} | i \in [k]\right\}\right) \setminus \{p_{n-k,\wr t \wr} | t \in [n-k-1]\}\\
&= \left(\left\{p_{n-k,j'_{k+1}}; \dots; p_{n-k,j'_{n-1}}\right\} \cup \left\{\overline{p_{n,j'_1}}; \dots; \overline{p_{n-k+1,j'_k}}\right\}\right) \setminus \{p_{n-k,\wr 1 \wr}; \dots; p_{n-k,\wr n-k-1 \wr}\}
\end{aligned}$$

Now, the first and the third curly brackets are just another expressions for the same set of clauses and $j'_k = j_k$, hence $c_2 = \left\{\overline{p_{n,j_1}}; \dots; \overline{p_{n-k+2,j_{k-1}}}; \overline{\mathbf{p_{n-k+1,j_k}}}\right\}$.

We can clearly see, that $c_1, c_2 \vdash_R c_3$.

- *sum-v2*, $a_3 = \left(\vec{x} = (j_1, \dots, j_{k-1}, j_k \equiv s, 0, \dots), \vec{y} = (z+1, \dots)\right)$.

$$
\begin{aligned}
&a_1 \equiv \left(\vec{x} = (j_1, \dots, j_k \equiv s, 0, \dots), \vec{y} = (z, \dots)\right)\\
&a_2 \equiv \left(\vec{x} = (j_1, \dots, j_{k-1}, \wr z+1 \wr, s, 0. \dots), \vec{y} = (n-k-1, \dots)\right)\\
&\vdash c_3 \equiv \left(\left(P^R_{n-k+1} \backslash P^{\{j_1,\dots,j_{k-1}\}}_{n-k+1}\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [k-1]\}\right) \setminus \{p_{n-k+1,\wr t \wr} | t \in [z+1]\}\\
&= \left\{p_{n-k+1,j_{k+z+1}}; \dots; p_{n-k+1,j_{n-1}}; \overline{p_{n,j_1}}; \dots; \overline{p_{n-k+2,j_{k-1}}}\right\}\\
&c_1 = \left(\left(P^R_{n-k+1} \backslash P^{\{j_1,\dots,j_{k-1}\}}_{n-k+1}\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [k-1]\}\right) \setminus \{p_{n-k+1,\wr t \wr} | t \in [z]\}\\
&= \left\{\mathbf{p_{n-k+1,j_{k+z}}}; \dots; p_{n-k+1,n-1}; \overline{p_{n,j_1}}; \dots; \overline{p_{n-k+2,j_{k-1}}}\right\}
\end{aligned}
$$

Note, that $j_{k+z} = \wr z+1 \wr$ ($\wr z+1 \wr$ computed as from address $(j_1, \dots, j_{k-1}, \wr z+1 \wr)$).

For c_2 we proceed as in $sum - v1.c_2$ - we again join cases $a_2.\vec{y}[1] = 1$, $a_2.\vec{y}[1] > 1$ and introduce notation $(j'_1, \dots, j'_k, s)$ for $(j_1, \dots, j_{k-1}, \wr z+1 \wr, s)$.

$$
\begin{aligned}
&c_2 = \left(\left(P^R_{n-k} \backslash P^{\left\{j'_1 \dots, j'_{k-1}, j'_k\right\}}_{n-k}\right) \cup \left\{\overline{p_{n-i+1,j'_i}} | i \in [k]\right\}\right) \setminus \{p_{n-k,\wr t \wr} | t \in [n-k-1]\}\\
&= \left\{\overline{p_{n-i+1,j'_1}}; \dots; \overline{\mathbf{p_{n-k+1,j'_k}}}\right\}
\end{aligned}
$$

Now because $j'_k = \wr z+1 \wr$, $c_1, c_2 \vdash_R c_3$ holds.

- *sum-v3*, $a_3 = \left(\vec{x} = (j_1, \dots, j_{n-2}, \wr 2 \wr \equiv s, 0, \dots), \vec{y} = (1, \dots)\right)$.

$$
\begin{aligned}
&a_1 \equiv \left(\vec{x} = (j_1, \dots, j_{n-2}, 0, \dots), \vec{y} = (2, \dots)\right)\\
&a_2 \equiv \left(\vec{x} = (j_1, \dots, j_{n-2}, \wr 1 \wr, 0), \vec{y} = (1, \dots)\right)\\
&\vdash c_3 \equiv \left(\left(P^R_2 \backslash P^{\{j_1,\dots,j_{n-2}\}}_2\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [n-2]\}\right) \setminus \{p_{2,\wr 1 \wr}\}\\
&= \left\{\overline{p_{n,j_1}}; \dots; \overline{p_{3,j_{n-2}}}\right\}\\
&c_1 = \left(P^R_2 \backslash \left(P^{\{j_1,\dots,j_{n-3}\}} \cup \left\{p_{2,j_{n-2}}\right\}\right)\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [n-2]\}\\
&= \left\{\mathbf{p_{2,j_{n-1}}}\right\} \cup \left\{\overline{p_{n,j_1}}; \dots; \overline{p_{3,j_{n-2}}}\right\}\\
&c_2 = \left(P^R_1 \backslash \left(P^{\{j_1,\dots,j_{n-2}\}}_1 \cup \left\{p_{1,j_{n-1} \equiv \wr 1 \wr}\right\}\right)\right) \cup \{\overline{p_{n-i+1,j_i}} | i \in [n-1]\}\\
&= \left\{\overline{p_{n,j_1}}; \dots; \overline{\mathbf{p_{2,j_{n-1}}}}\right\}
\end{aligned}
$$

It is easy to see, that $c_1, c_2 \vdash_R c_3$.

- *sum-v4*, $a_3.\vec{x} = (j_1 \equiv s, 0, \dots)$.

Defining $a = (\vec{x} = (0, \dots), \vec{y} = (i, \dots))$ as $PHP_1\left([n], [n-1]\right)[i] = P^R_i$ is correct, as the formula c_2 becomes exactly P^R_i. Thus the current case *sum-v4* is covered by the previous cases *sum-v1/2*.

Note that nowhere in the proof we have used that PHP is, in fact, a tautology.

□

Bibliography

[1] Krajíček, J.: *Implicit proofs.* The Journal of Symbolic Logic, vol. 69 (2004), 387-397.

[2] Cook A., Reckhow R.A.: *The relative efficiency of propositional proof systems.* Journal of Symbolic Logic, vol. 44 (1979), 36-50.

[3] Cook, S.A.: *The complexity of the theorem proving procedures.* Proceedings of the Third Annual ACM Symposium on the Theory of Computing, 1971, 151-158.

[4] Reckhow, R. A.: *On the lengths of proofs in the propositional calculus.* PhD thesis, Dept. of CS, University of Toronto, Technical Report No.87, 1976.

[5] Blake, A.: *Canonical expressions in Boolean algebra.* PhD thesis, University of Chicago, 1937.

[6] Haken, A.: *The intractability of resolution.* Theoretical Computer Science, vol. 39 (1985), 297-308.

[7] Buss, S.R.: *Polynomial size proofs of the propositional pigeonhole principle.* The Journal of Symbolic Logic, vol. 52 (1987), 916-927.