IMPLEMENTING DEDUCTION CHAINS FOR THE PROOF SEARCH CALCULUS

BACHELOR'S THESIS

PRESENTED TO THE INSTITUTE OF COMPUTER SCIENCE AND APPLIED MATHEMATICS OF THE UNIVERSITY OF BERN IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF SCIENCE

PETAR BOGDANOVIĆ

Advised by
Prof. Dr. Gerhard Jäger

BERN, JUNE 2015

Contents

Abstract

Within the formal system that is the proof search calculus, the method of building deduction chains is an algorithmic way to prove or disprove the validity of well formed formulas in the classical propositional logic. This document reviews the proof search calculus, its language and deductive system and finally describes a sample implementation that builds deduction chains in order to automatically check the validity of user-defined input.

Chapter 1

Classical Propositional Logic

One of the elements that make up the proof search calculus (PSC) is its underlying language, the classical propositional logic (CP). This chapter will review symbols, syntax and semantics of CP and define what it means for well formed formulas (in CP) to be *satisfiable*, *valid* and *equivalent*.

1.1 Symbols

The basic building blocks of CP are *atomic propositions*, i.e. statements that can stand on their own and are not the result of composing statements. Out of the (countably infinite) set of atomic propositions, more complex compound statements are built with *logical connectives*.

The set of all declared symbols in CP is the union of the following sets:

- atomic propositions: $\{p_i \colon i \in \mathbb{N}\}$
- logical connectives: $\{\top, \bot, \neg, \wedge, \vee, \rightarrow\}$

$\top$	(verum)
$\bot$	(falsum)
$\neg$	(negation)
$\wedge$	(conjunction)
$\vee$	(disjunction)
$\rightarrow$	(implication)

The logical connectives $\{\top, \bot\}$ are also called *propositional constants*, often defined as constant functions that take no arguments.

1.2 Syntax

Formulas in CP are inductively defined as follows:

1. Every atomic proposition is a formula.
2. Every propositional constant is a formula.
3. If F and G are formulas, then $(\neg F)$, $(F \wedge G)$, $(F \vee G)$ and $(F \to G)$ are formulas.

When compiling formulas, brackets may be omitted whenever there is no danger of confusion. It is assumed that negation $(\neg)$ has precedence over other connectives [Met14].

1.3 Semantics

Atomic propositions and, by extension, formulas have no meaning as long as no *valuation* exists that assigns values to them.

A valuation V maps atomic propositions (*atoms*) to truth values:

$$V\colon \{p_i\colon i \in \mathbb{N}\} \mapsto \{\mathbf{t}, \mathbf{f}\}$$

$\bar{V}$ extends V by mapping formulas to truth values. If F and G are formulas, $\bar{V}$ is inductively defined as follows:

$$\begin{aligned}
\bar{V}(p_i) &= V(p_i) \\
\bar{V}(\top) &= \mathbf{t} \\
\bar{V}(\bot) &= \mathbf{f} \\
\bar{V}(\neg F) &= \begin{cases} \mathbf{t} & \bar{V}(F) = \mathbf{f} \\ \mathbf{f} & \text{otherwise} \end{cases} \\
\bar{V}(F \wedge G) &= \begin{cases} \mathbf{t} & \bar{V}(F) = \bar{V}(G) = \mathbf{t} \\ \mathbf{f} & \text{otherwise} \end{cases} \\
\bar{V}(F \vee G) &= \begin{cases} \mathbf{t} & \bar{V}(F) = \mathbf{t} \text{ or } \bar{V}(G) = \mathbf{t} \\ \mathbf{f} & \text{otherwise} \end{cases} \\
\bar{V}(F \to G) &= \begin{cases} \mathbf{t} & \bar{V}(F) = \mathbf{f} \text{ or } \bar{V}(G) = \mathbf{t} \\ \mathbf{f} & \text{otherwise} \end{cases}
\end{aligned}$$

From this definition it follows that the truth value of a formula is uniquely determined by the truth values of the atoms that occur in it [Met14].

1.4 Satisfiability, Validity and Equality

Valuations may characterise formulas and reveal relations between them:

- A formula F is *satisfiable* if there exists a valuation V so that $\bar{V}(F) = \mathbf{t}$.
- A formula F is *valid* if $\bar{V}(F) = \mathbf{t}$ for all valuations V.
- Formulas F and G are *equivalent* if $\bar{V}(F) = \bar{V}(G)$ for all valuations V.

1.5 Negation Normal Form

In order to reduce the structural complexity of formulas, the proof search calculus requires formulas to be in *negation normal form* (NNF). In NNF, there are no implications ($\rightarrow$) or consecutive negations ($\neg\neg$) and negations are generally only allowed to occur immediately in front of atoms. The remaining connectives $\{\wedge, \vee, \neg\}$ are still functionally complete [Met14] and every formula in CP is equivalent to its NNF counterpart [JP14].

Formulas in NNF are inductively defined as follows:

1. All atoms and their negations are in NNF.
2. Propositional constants (but not their negations) are in NNF.
3. If F and G are in NNF, then $(F \wedge G)$ and $(F \vee G)$ are in NNF.

The process of transforming formulas (in CP) into their negation normal form makes use of the following equivalences:

$$\begin{aligned}
(F \rightarrow G) &\sim (\neg F \vee G) \\
\neg(F \wedge G) &\sim (\neg F \vee \neg G) \\
\neg(F \vee G) &\sim (\neg F \wedge \neg G) \\
\neg\neg F &\sim F \\
\neg\top &\sim \bot \\
\neg\bot &\sim \top
\end{aligned}$$

Given the algorithm nnf and a formula A, $\text{nnf}(A)$ can be carried through with time-complexity polynomial in $\ell(A)$, where $\ell(A)$ is the number atoms and connectives occurring in A [JP14].

Chapter 2

Proof Search Calculus

Testing formulas in negation normal form for validity is hampered by the fact that a disjunction $(\vee)$ can be valid even if none of its operands are valid $(P \vee \neg P)$. In order to deal with this case in a systematic way, PSC derives *finite sequences* of formulas instead of individual formulas.

2.1 Finite Sequences of Formulas

Finite sequences of formulas in NNF (abbreviated as *sequences* and not to be confused with *sequences of sequences*) are denoted by capital Greek letters $(\Gamma, \Delta, \Pi, \Sigma)$ and defined as follows:

$$\Gamma = A_1, ..., A_n$$

Sequences can be interpreted disjunctively:

$$\Gamma^{\vee} = A_1 \vee ... \vee A_n$$

Valuations always interpret sequences disjunctively:

$$\bar{V}(\Gamma) = \bar{V}(\Gamma^{\vee}) = \bar{V}(A_1 \vee ... \vee A_n)$$

Intuitively, sequences can be described as links in the *sequence of sequences* that is the proof in PSC.

2.2 Proof in PSC

A proof in PSC is a finite *sequence of sequences* $\Gamma_0, ..., \Gamma_n$ where Γ_i is either an axiom in PSC or the conclusion of a rule, while all premises of the rule occur left of Γ_i. A sequence Γ is derivable in PSC $(\vdash_{PSC} \Gamma)$ if there exists a proof in PSC whose last member is Γ.

2.3 Axioms in PSC

For all sequences Γ, Δ, Π and all formulas A which are either an atomic proposition or the negation of an atomic proposition, the following axioms are defined:

$$\Gamma, \top, \Pi \tag{True}$$

$$\Gamma, A, \Delta, \overline{A}, \Pi \tag{Id}$$

Where the whole axiom is *one* sequence, $\overline{A}$ is the complement of A and the complement itself is defined as follows:

$$\overline{A} = \neg A$$
$$\overline{\neg A} = A$$

2.4 Rules of Inference in PSC

For all sequences Γ, Δ and all formulas A, B, the following two rules of inference are defined:

$$\frac{\Gamma, A, B, \Delta}{\Gamma, A \vee B, \Delta} \tag{$\vee$}$$

$$\frac{\Gamma, A, \Delta \qquad \Gamma, B, \Delta}{\Gamma, A \wedge B, \Delta} \tag{$\wedge$}$$

Where the premises of the rule appear above and the conclusion of the rule below the line.

2.5 Sample Proof

For all atomic formulas A and axiom 15 (A15) of the Hilbert Calculus, the following proof can be derived in PSC:

$$\Gamma = (A \rightarrow (A \wedge \neg A)) \rightarrow \neg A \tag{A15}$$

$$\Gamma_0 = \neg A, A, \neg A \tag{axiom}$$

$$\Gamma_1 = (\neg A \vee A), \neg A \tag{$\vee$: 0}$$

$$\Gamma_2 = A, \neg A \tag{axiom}$$

$$\Gamma_3 = A \wedge (\neg A \vee A), \neg A \tag{$\wedge$: 1,2}$$

$$\Gamma_4 = (A \wedge (\neg A \vee A)) \vee \neg A \tag{$\vee$: 3}$$

Since Γ_4 is the negation normal form of Γ, we may write $\vdash_{PSC} \Gamma$ and given that the axioms in PSC are valid and the rules of inference in PSC preserve validity [JP14], we may also write $\models \Gamma$ or, literally, *A15 is valid*.

2.6 Deduction Chains

Building deduction chains is a more systematic way of proving formulas in PSC. Instead of starting with axioms and applying inference rules towards a particular formula, we start with the formula and proceed inversely to the inference rules.

The deduction chain generating algorithm introduces three additional characteristics of formulas and sequences:

1. A formula is *irreducible* if it is an atom, the negation of an atom, or a propositional constant. All other formulas are *reducible*.
2. A sequence that contains at least one reducible formula is *reducible*. All other sequences are *irreducible*.
3. The right most reducible formula of a reducible sequence is the *distinguished* formula of that sequence.

A deduction chain for a sequence Γ is a sequence of sequences $\Gamma_1, \Gamma_2, \Gamma_3, ...$ generated as follows [JP14]:

(D1) The initial sequence Γ_1 of the deduction chain is the sequence Γ.

(D2) If the sequence Γ_n of the deduction chain is *irreducible* or an axiom in PSC, then it is the last sequence of this deduction chain.

(D3) If the sequence Γ_n of the deduction chain is *reducible* and not an axiom in PSC, then Γ_n has an immediate successor Γ_{n+1} in this deduction chain. Given that Γ_n is reducible, it contains a *distinguished* formula A and therefore can be written as

$$\Pi_n, A, \Sigma_n.$$

Based on A, we then distinguish the following two cases:

(a) A is the formula $B \vee C$. Then Γ_{n+1} is the sequence

$$\Pi_n, B, C, \Sigma_n.$$

(b) A is the formula $B \wedge C$. Then Γ_{n+1} is the sequence

$$\Pi_n, B, \Sigma_n \quad \text{or} \quad \Pi_n, C, \Sigma_n.$$

If *every* deduction chain for Γ ends with an axiom in PSC, then there is proof for Γ in PSC ($\vdash_{PSC} \Gamma$). If, on the other hand, there exists a deduction chain that does *not* end with an axiom in PSC, then there exists a valuation V such that $\bar{V}(\Gamma) = \mathrm{f}$ [JP14].

2.7 Sample Proof #2

Below are all deduction chains for a sample input Γ:

$$\Gamma = ((A \land \top) \lor C) \lor (\neg C \land (B \lor \bot))$$

$$\begin{aligned}
\Gamma_1 &= \Gamma && \text{(D1)}\\
\Gamma_2 &= (A \land \top) \lor C,\ \neg C \land (B \lor \bot) && \text{(D3-a)}\\
\Gamma_3 &= (A \land \top) \lor C,\ \neg C && \text{(D3-b)}\\
\Gamma_4 &= (A \land \top),\ C,\ \neg C && \text{(D3-a, }\textbf{axiom}\text{)}
\end{aligned}$$

$$\begin{aligned}
\Gamma_1' &= \Gamma_1 && \\
\Gamma_2' &= \Gamma_2 && \\
\Gamma_3' &= (A \land \top) \lor C,\ B \lor \bot && \text{(D3-b)}\\
\Gamma_4' &= (A \land \top) \lor C,\ B,\ \bot && \text{(D3-a)}\\
\Gamma_5' &= (A \land \top),\ C,\ B,\ \bot && \text{(D3-a)}\\
\Gamma_6' &= A,\ C,\ B,\ \bot && \text{(D3-b, }\textbf{irreducible}\text{)}
\end{aligned}$$

$$\begin{aligned}
\Gamma_1'' &= \Gamma_1' && \\
\Gamma_2'' &= \Gamma_2' && \\
\Gamma_3'' &= \Gamma_3' && \\
\Gamma_4'' &= \Gamma_4' && \\
\Gamma_5'' &= \Gamma_5' && \\
\Gamma_6'' &= \top,\ C,\ B,\ \bot && \text{(D3-b, }\textbf{axiom}\text{)}
\end{aligned}$$

Since one deduction chain ends with a sequence that is no axiom (Γ_6'), there exists a valuation V such that $\bar{V}(\Gamma) = \mathrm{f}$:

$$\bar{V}_{(A|\mathrm{f},\, C|\mathrm{f},\, B|\mathrm{f})}(\Gamma) = \mathrm{f}$$

Therefore, there is no proof for Γ in PSC and Γ is not valid.

Chapter 3

Implementation

Equipped with the theoretical foundation of the proof search calculus, we are now able to design and implement a simple application that translates an arbitrary sequence of formulas into a set of deduction chains.

3.1 Subproblems

The problem of building deduction chains out of user defined sequences can be split into several (mostly) non-overlapping subproblems:

- Parse a list of user defined formulas.
- Build syntax tree for each formula.
- Negation-normalize all syntax trees.
- Group formulas into one sequence.
- Derive all deduction chains from sequence.

3.2 Parser

A parser grammar capable of accepting CP must implement the following:

- One unary operator $\{\neg\}$.
- Three binary operators $\{\rightarrow, \wedge, \vee\}$.
- Operator precedence $(\neg, \wedge, \vee, \rightarrow)$.

Since no two operators have the same precedence, additional rules regarding operator associativity are not required.

3.3 Syntax Tree

A regular binary tree is suitable for representing all operations. Each child is an operand (or another operation) and does not need to know its parent. The negation ($\neg$) only defines one child.

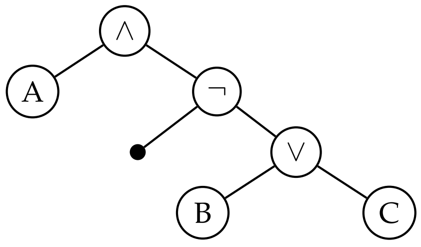


Figure 3.1: Tree representation of $(A \land \neg(B \lor C))$.

3.4 Negation Normal Form

Negation-normalizing a syntax tree is marginally more complex than a tree traversal. Implications need to be resolved first. Then all negations that are not immediately preceding atoms need to be be pushed downwards.

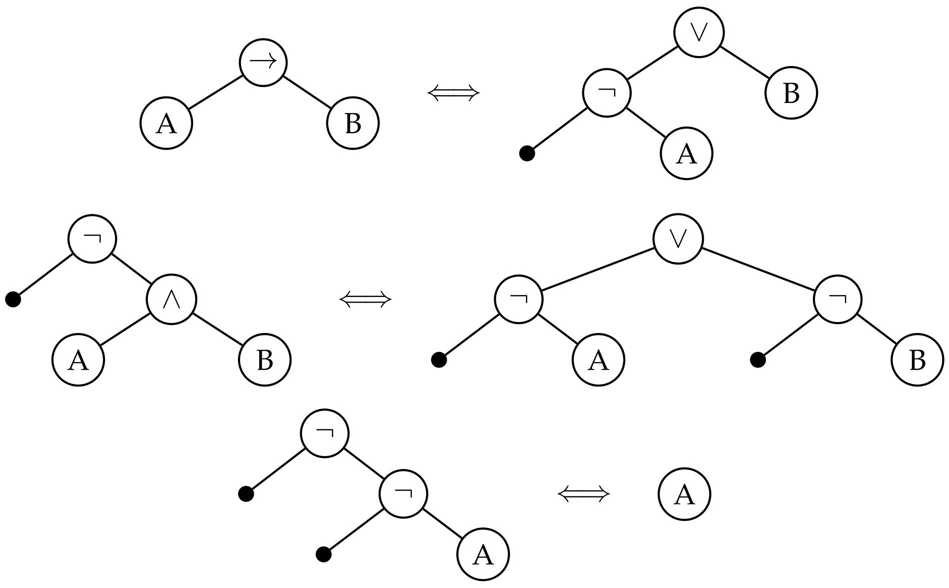


Figure 3.2: Tree transformation examples.

3.5 Sequences and Chains

A significant part of the process of generating deduction chains involves iterating over a number of sequences in order to find out if they are *axioms*, *reducible* or *irreducible*. Simple arrays of formulas are well suited for that purpose even though some efficiency gains could be achieved with more complex data structures.

When generating deduction chains, conjunctions in sequences create branching points, where two immediate successors are possible. This characteristic suggests that all deduction chains can be represented by a binary tree.

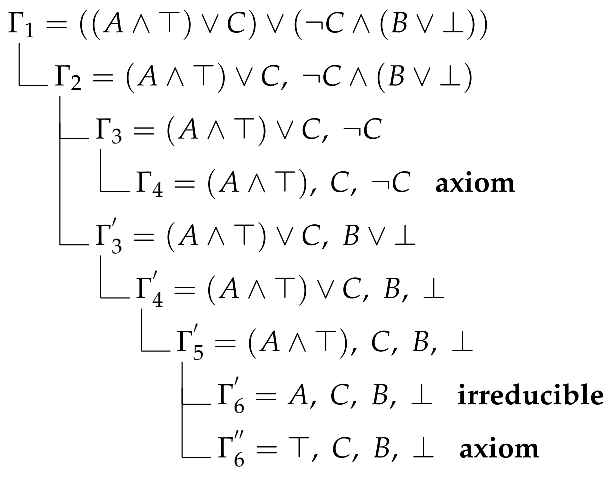

$\Gamma_1 = ((A \wedge \top) \vee C) \vee (\neg C \wedge (B \vee \bot))$
$\Gamma_2 = (A \wedge \top) \vee C,\ \neg C \wedge (B \vee \bot)$
$\Gamma_3 = (A \wedge \top) \vee C,\ \neg C$
$\Gamma_4 = (A \wedge \top),\ C,\ \neg C$ **axiom**
$\Gamma_3' = (A \wedge \top) \vee C,\ B \vee \bot$
$\Gamma_4' = (A \wedge \top) \vee C,\ B,\ \bot$
$\Gamma_5' = (A \wedge \top),\ C,\ B,\ \bot$
$\Gamma_6' = A,\ C,\ B,\ \bot$ **irreducible**
$\Gamma_6'' = \top,\ C,\ B,\ \bot$ **axiom**

Figure 3.3: Revisited representation of proof in Section 2.7.

3.6 Complexity

Given an input that is a disjunction of conjunctive clauses of two literals (where a *literal* is an atom or the negation of an atom), the worst-case time complexity of a deduction chain generating algorithm is exponential in the number of conjunctive clauses. The example

$$(A \wedge B) \vee (C \wedge D) \vee (E \wedge F) \vee (G \wedge H)$$

will yield 2^4 or 16 deduction chains, since it contains 4 conjunctive clauses.

3.7 Chainer

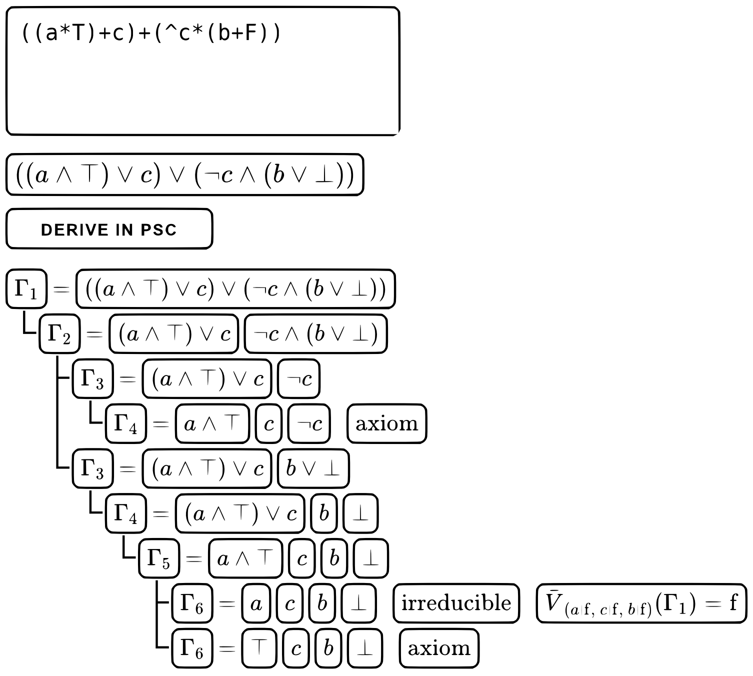
Chainer is a sample implementation of a deduction chain generating algorithm in client-side JavaScript. One or more formulas can be entered into a simple form and subsequently transformed into a set of deduction chains.

Chainer accepts the following simplified set of symbols:

- atomic propositions $\{a, b, c, \ldots, x, y, z\}$
- logical connectives $\{\neg, \wedge, \vee, \rightarrow\}$ or {`^`,`*`,`+`,`->`}
- propositional constants $\{\top, \bot\}$ or {`T`,`F`}

Ambiguous statements are interpreted by enforcing a given precedence of operators $(\neg, \wedge, \vee, \rightarrow)$ and assuming right associativity. Manually inserted parentheses may override both assumptions.

Changes to the input statements automatically trigger the procedure of parsing all formulas and converting them into their negation normal form. If the parsed output looks acceptable, *derive in PSC* will start generating all deduction chains. All leaves of the resulting tree will be marked according to section 2.7 where *irreducible* leaves will also contain the description of a specific valuation V so that $\bar{V}(\Gamma) = \mathbf{f}$.



```
((a*T)+c)+(^c*(b+F))
```

$((a \wedge \top) \vee c) \vee (\neg c \wedge (b \vee \bot))$

DERIVE IN PSC

$\Gamma_1 = ((a \wedge \top) \vee c) \vee (\neg c \wedge (b \vee \bot))$
$\quad\Gamma_2 = (a \wedge \top) \vee c \quad \neg c \wedge (b \vee \bot)$
$\quad\quad\Gamma_3 = (a \wedge \top) \vee c \quad \neg c$
$\quad\quad\quad\Gamma_4 = a \wedge \top \quad c \quad \neg c$ axiom
$\quad\quad\Gamma_3 = (a \wedge \top) \vee c \quad b \vee \bot$
$\quad\quad\quad\Gamma_4 = (a \wedge \top) \vee c \quad b \quad \bot$
$\quad\quad\quad\quad\Gamma_5 = a \wedge \top \quad c \quad b \quad \bot$
$\quad\quad\quad\quad\quad\Gamma_6 = a \quad c \quad b \quad \bot$ irreducible $\bar{V}_{(a|\mathrm{f},\, c|\mathrm{f},\, b|\mathrm{f})}(\Gamma_1) = \mathrm{f}$
$\quad\quad\quad\quad\quad\Gamma_6 = \top \quad c \quad b \quad \bot$ axiom

Figure 3.4: Screenshot of `https://smokva.net/chainer/`.

3.7.1 Design

Chainer is divided into three logical modules:

- The *Parser* module contains the language grammar, the Syntax `Tree` object, the negation normalizing functions and other helpers that act on `Tree` objects or simply describe their properties.
- The *Chainer Library* introduces the additional abstract objects `Formula`, `Sequence` and `Chain`. While `Formula` is merely a verbose extension of the `Tree` object, the functionality associated with `Sequence` describes a major part of the deduction chain generating algorithm.
- The *User Interface* describes all components of the HTML user interface and the underlying logic that manages the scene and issues calls to the Chainer Library. Additionally, this module introduces several functions that generate visual representations of abstract objects from the Chainer Library.

3.7.2 Parsing Expression Grammar

The CP formula parser used by Chainer is generated from a simple *parsing expression grammar* (PEG). PEGs cannot be ambiguous; in the case of more than one way to match an expression, the first match is chosen. Rules may not be left recursive [For04].

Chainer defines the following PEG, written in PEG.js [Maj15]:

```
start = implication

implication
    = left:disjunction "->" right:implication
        { return new Tree(SYMB_IMPL,
            TYPE_LCON, undefined, left, right) }
    / left:disjunction "→" right:implication
        { return new Tree(SYMB_IMPL,
            TYPE_LCON, undefined, left, right) }
    / disjunction

disjunction
    = left:conjunction [+∨] right:disjunction
        { return new Tree(SYMB_DISJ,
            TYPE_LCON, undefined, left, right) }
    / conjunction
```

```
conjunction
    = left:negation [*∧] right:conjunction
        { return new Tree(SYMB_CONJ,
            TYPE_LCON, undefined, left, right) }
    / negation

negation
    = [¬^] right:primary
        { return new Tree(SYMB_NEGT,
            TYPE_LCON, undefined, undefined, right) }
    / [¬^] right:negation
        { return new Tree(SYMB_NEGT,
            TYPE_LCON, undefined, undefined, right) }
    / primary

primary
    = [T⊤]
        { return new Tree(SYMB_TRUE,
            TYPE_PROP, TYPE_CONS, undefined, undefined) }
    / [F⊥]
        { return new Tree(SYMB_FALS,
            TYPE_PROP, TYPE_CONS, undefined, undefined) }
    / symb:[a-z]
        { return new Tree(symb,
            TYPE_PROP, TYPE_ATOM, undefined, undefined) }
    / "(" symb:implication ")"
        { return symb }
```

3.7.3 Syntax Tree Object

The `Tree` constructor has the following parameters:

1. `symb`: Root node symbol; any allowed atomic proposition (string) or one of the following predefined symbols:

    - `SYMB_IMPL` ("→" or "->")
    - `SYMB_DISJ` ("∨" or "+")
    - `SYMB_CONJ` ("∧" or "*")
    - `SYMB_NEGT` ("¬" or "^")
    - `SYMB_TRUE` ("⊤" or "T")
    - `SYMB_FALS` ("⊥" or "F")

2. `type`: Root node symbol type; one of the predefined types `TYPE_LCON` (logical connective) or `TYPE_PROP` (proposition).

3. `subtype`: Root node symbol subtype; one of the predefined `TYPE_PROP` subtypes, `TYPE_ATOM` (atomic) or `TYPE_CONS` (constant). `TYPE_LCON` has no subtypes.

4. `left`: Left child; any allowed atomic proposition (string), `SYMB_TRUE`, `SYMB_FALS` or another `Tree` object.

5. `right`: Right child (prefered when root node is unary operation); any allowed atomic proposition (string), `SYMB_TRUE`, `SYMB_FALS` or another `Tree` object.

`Tree` objects have the following properties:

- `Tree.symb`: Root node symbol.
- `Tree.type`: Root node symbol type.
- `Tree.subtype`: Root node symbol subtype.
- `Tree.left`: Left child.
- `Tree.right`: Right child.

The following functions operate on `Tree` objects:

- `Tree.walk()`: Returns human readable formula represented by `Tree`.
- `Tree.walk_tex()`: Returns formula represented by `Tree` in TEX. This function uses the predefined conversion table `SYMB_TO_TEX`.
- `Tree.nnf()`: Negation normalizes `Tree`.
- `Tree.nnf_impl()`: Implication reduction, called by `Tree.nnf()`.
- `Tree.nnf_negt()`: Negation normalization, called by `Tree.nnf()`.
- `Tree.is_reducible()`: Returns `true` if formula represented by `Tree` is reducible, `false` otherwise.
- `Tree.split()`: Returns object if `Tree` is reducible, `undefined` otherwise. The returned object has the following properties:
  - `left`: Left child of `Tree`.
  - `right`: Right child of `Tree`.
  - `disj`: `true` if `Tree` represents a disjunction, `false` otherwise.

- `Tree.get_root_prop()`: Returns object if `Tree` is a literal, `undefined` otherwise. The returned object has the following properties:
  - `prop`: Root node symbol of `Tree` if `Tree` represents an atomic proposition, `undefined` otherwise.
  - `prop_negt`: Symbol in `Tree.right` if `Tree` represents the negation of an atomic proposition, `undefined` otherwise.
- `Tree.root_is_true()`: Returns `true` if `Tree` represents propositional constant *verum* ($\top$), `false` otherwise.

3.7.4 Formula Object

The `Formula` constructor has one parameter: the `input` string. Based on the parsing results of `input`, one or more of the following `Formula` properties will be set:

- `Formula.input_raw`: Raw `input`.
- `Formula.input`: `input` without whitespace characters.
- `Formula.empty`: `true` if `input` was empty.
- `Formula.tree_raw`: Non-normalized `Tree` representing `input`.
- `Formula.error`: Errors encountered during parsing of `input`.
- `Formula.broken`: `true` if parsing of `input` failed.
- `Formula.tree`: Negation normalized `Tree` representing `input`.
- `Formula.output_raw`: Return value of `Formula.tree_raw.walk()`.
- `Formula.output`: Return value of `Formula.tree.walk()`.
- `Formula.output_tex`: Return value of `Formula.tree.walk_tex()`.

The following functions operate on `Formula` objects:

- `Formula.clone()`: Returns copy of `Formula`.

3.7.5 Sequence Object

The `Sequence` constructor has an array of input strings (`fms`) as parameter. Based on the parsing results of all elements, one or more of the following `Sequence` properties will be set:

- `Sequence.fms`: Array of `Formula` objects built from `fms` elements.

- `Sequence.broken`: `true` if parsing of one or more `fms` elements failed.
- `Sequence.empty`: `true` if `fms` had no elements.

The following functions operate on `Sequence` objects:

- `Sequence.is_axiom()`: Returns `true` if `Sequence` is an axiom.
- `Sequence.is_reducible()`: Returns `true` if `Sequence` contains one or more reducible `Formula` objects.
- `Sequence.get_dist_form()`: Returns array index of the distinguished `Formula` if `Sequence` is reducible, `undefined` otherwise.
- `Sequence.get_successor()`: Returns array if `Sequence` is reducible, `undefined` otherwise. The returned array has the following elements:
  0. Succeeding `Sequence`.
  1. Alternate succeeding `Sequence` in case the current distinguished `Formula` represents a conjunction, `undefined` otherwise.
- `Sequence.get_valuation()`: Returns object if `Sequence` is *irreducible*, `undefined` otherwise. The returned object maps one or more atoms to truth values and therefore defines valuation V so that $\bar{V}(\Gamma) = \mathbf{f}$.
- `Sequence.get_valuation_tex()`: Returns valuation V in $\TeX$.

3.7.6 Chain Object

The `Chain` object is a binary tree that holds a sequence as its root node and zero, one or two sequences as its children. The term *Chain* itself is obviously a misnomer and *ChainTree* would probably have been a better choice.

The `Chain` constructor has a `Sequence` object (`seq`) as parameter. Based on the characteristics of `seq`, one or more of the following `Chain` properties will be set:

- `Chain.broken`: `true` if `seq` is broken.
- `Chain.root`: Root node `Sequence`; set to `seq`.
- `Chain.axiom`: `true` if root node `Sequence` is an axiom.
- `Chain.left`: `Chain` object containing succeeding `Sequence` in case root node `Sequence` has one, `undefined` otherwise.
- `Chain.right`: `Chain` object containing alternate succeeding `Sequence` in case root node `Sequence` has one, `undefined` otherwise.

The following functions operate on `Chain` objects:

- `Chain.log()`: Prints all deduction chains by traversing `Chain`.

With `Chain` defined, generating a list of deduction chains for any formula now takes less than five lines of code:

```
> var f = "((a*T)+c)+(^c*(b+F))"
> var s = new Sequence([f])
> var c = new Chain(s)
> c.log()

[{((a * T) + c) + (^c * (b + F))}]
[{(a * T) + c}{^c * (b + F)}]
[{(a * T) + c}{^c}]
[{a * T}{c}{^c}] (axiom)

[{((a * T) + c) + (^c * (b + F))}]
[{(a * T) + c}{^c * (b + F)}]
[{(a * T) + c}{b + F}]
[{(a * T) + c}{b}{F}]
[{a * T}{c}{b}{F}]
[{a}{c}{b}{F}] (irreducible)

[{((a * T) + c) + (^c * (b + F))}]
[{(a * T) + c}{^c * (b + F)}]
[{(a * T) + c}{b + F}]
[{(a * T) + c}{b}{F}]
[{a * T}{c}{b}{F}]
[{T}{c}{b}{F}] (axiom)
```

3.7.7 User Interface

Chainer uses the React JavaScript library [Fac15] for its more intuitive user interface captured in figure 3.4. The resulting single-page application uses the following React components:

- `Editor`: Container for text input area and `<Formulas>`.
- `Formulas`: Container for one or more `<Formula>` and `<Chains>`.
- `Formula`: A successfully parsed formula or a parser error message.
- `Chains`: Container for *derive in..* button and `<ChainsTree>`.

- `ChainsTree`: A tree-like construct representing all deduction chains.

In order to efficiently translate any `Chain` object into a nested list contained by `<ChainsTree>`, the user interface module extends `Chain` by the function `get_li()`. `Chain.get_li()` traverses `Chain` and returns a nested list that can be rendered without any additional post-processing.

3.7.8 Installation

The Chainer source package contains all dependencies except *Node.js*, the JavaScript interpreter required to build Chainer. In order to run an instance of Chainer in a Debian environment, the following steps are neccessary:

```
$ sudo apt-get install git nodejs-legacy
$ git clone https://github.com/petarb/chainer
$ cd chainer
$ make
$ firefox index.html
```

Similar procedure in an OSX environment containing MacPorts:

```
$ sudo port install git nodejs
$ git clone https://github.com/petarb/chainer
$ cd chainer
$ make
$ open index.html
```

Publishing Chainer through a web server (or other channels) is a matter of recursively copying (or transfering) two elements:

```
$ cp -R index.html modules /path/to/www
```

Bibliography

Fac15 Facebook. `http://facebook.github.io/react/`, 2015.

For04 Bryan Ford. Parsing Expression Grammars, 2004.

JP14 G. Jäger and D. Probst. Discrete Mathematics and Logic, 2014.

Maj15 David Majda. `http://pegjs.org/documentation/`, 2015.

Met14 George Metcalfe. Mathematical Logic and Model Theory, 2014.