Automatic Construction of Fitch Proofs with Correct Line References

A Technical Guide for $\text{fitch}_{\text{printer.pl}}$ and $\text{latex}_{\text{utilities.pl}}$

[2025-08-02 sam.]

Contents

1 Introduction

This document provides a comprehensive guide to the automatic construction of Fitch-style natural deduction proofs with correct line references. The system implemented in `fitch_printer.pl` and `latex_utilities.pl` transforms G4 sequent calculus proofs into readable Fitch proofs with proper line numbering and rule justifications.

2 Core Concepts

2.1 The Line Tracking Problem

In Fitch proofs, every line must have:

1. A unique line number
2. Proper indentation (scope bars)
3. A formula
4. A justification referencing previous lines

The challenge is maintaining correct references across complex proof structures with nested assumptions and discharges.

2.2 Variable Tracking System

The current system uses several tracking variables that need clarification. We propose a uniform naming convention to replace Burse's cryptic variable names:

- `O` → `ScopeLevel`: Number of active assumptions (scope depth)
- `N` → `CurrentLine`: Current line number (input)
- `K` → `NextLine`: Next line number (output)
- `U` → `VarCounterIn`: Variable counter (input)
- `V` → `VarCounterOut`: Variable counter (output)
- `Context` → `ProofContext`: List of `LineNumber:Formula` pairs

2.2.1 Complete Uniform Variable Convention

Beyond the core variables, we establish a systematic naming convention for all variables used in proof construction:

Core Variables:

- `ScopeLevel`: Number of vertical bars to display (scope depth)
- `CurrentLine`: Line number of the last written line
- `NextLine`: Line number that will be assigned to the next line
- `CurrentPosition`: Synchronization position between parallel proof branches
- `VarCounterIn`: Input variable counter for formula rendering
- `VarCounterOut`: Output variable counter after formula rendering
- `TemporaryVarCounter`: Intermediate variable counter between processing steps
- `ProofContext`: List maintaining `LineNumber:Formula` mappings

Elimination Rule Variables (Prawitz Convention):

- `MajorPremissLine`: Line containing the main formula being eliminated
- `MinorPremissLine`: Line containing the minor premiss (for →E)
- `AssumptionA`, `AssumptionB`: Line numbers for case assumptions (disjunction elimination, etc.)
- `SubProofEndA`, `SubProofEndB`: End lines of subproofs for cases

2.2.2 ScopeLevel Definition and Behavior

`ScopeLevel` represents the number of vertical bars (`\fa`) to display in a Fitch line, corresponding to the depth of nested active assumptions.

Precise Rules:

1. `ScopeLevel = 0`: Main level line (no bars)
2. `ScopeLevel = 1`: One active assumption (one `\fa` bar)
3. `ScopeLevel = 2`: Two active assumptions (two `\fa \fa` bars)
4. `ScopeLevel` increases by 1 when making an assumption (`\fh`)
5. `ScopeLevel` decreases by 1 when discharging an assumption (introduction rules)

Important: `ScopeLevel` is never indexed - it is always a single scalar integer value passed between predicates. The value simply changes as we traverse the proof structure.

2.2.3 CurrentLine and NextLine Dynamics

These variables manage the sequential numbering of proof lines:

CurrentLine:

- Value: Line number of the last line actually written in the Fitch proof
- Role: Reference point for "where we are" in proof construction
- Dynamic: Changes constantly as lines are added

NextLine:

- Value: Line number that the next line will receive
- Calculation: Always `NextLine is CurrentLine + 1`
- Effect: After writing a line, `NextLine` becomes the new `CurrentLine` for subsequent steps

CurrentPosition:

- Value: Synchronization position between different proof branches or levels
- Role: Coordinates line numbering across parallel subproofs (especially in disjunction elimination)
- Distinction: Different from `CurrentLine` - used for inter-branch coordination

2.2.4 Variable Counter System

The variable counter system manages the generation of fresh variable names in formula rendering:

VarCounterIn: Input counter state when starting to process a formula or proof fragment **VarCounterOut:** Output counter state after processing, ensuring no variable name conflicts **TemporaryVarCounter:** Intermediate counter states between processing steps

3 Basic Rendering Primitives

3.1 Hypothesis Rendering

```
render_hypo(ScopeLevel, Formula, Label,
           CurrentLine, NextLine,
           VarCounterIn, VarCounterOut) :-
    NextLine is CurrentLine + 1,
    render_bars(ScopeLevel),
    write('\\fh '),
    rewrite(Formula, VarCounterIn, VarCounterOut,
           FormattedFormula),
    write(FormattedFormula),
    render_label(Label).
```

Example: Assuming A at line 3 with `ScopeLevel = 1`:

```
\fa \fh A & AS\\
```

3.2 Derived Line Rendering

```
render_have(ScopeLevel, Formula, Justification,
           CurrentLine, NextLine,
           VarCounterIn, VarCounterOut) :-
    NextLine is CurrentLine + 1,
    render_bars(ScopeLevel),
    rewrite(Formula, VarCounterIn, VarCounterOut,
           FormattedFormula),
    write(FormattedFormula),
    render_label(Justification).
```

Example: Deriving B from lines 1 and 2 at `ScopeLevel = 0`:

```
B & $\to E$ 1, 2\\
```

4 Context Management

The context maintains a mapping from line numbers to formulas:

```
Context = [3:A, 2:(A => B), 1:((A => B) => C)]
```

This allows the system to:

1. Find which line contains a specific formula
2. Verify that rule applications are valid
3. Generate correct line references in justifications

4.1 Safe Formula Lookup

```
find_formula_line_safe(Context, Formula, LineNumber) :-
    ( member(LineNumber:Formula, Context) ->
        true
    ; member(LineNumber:ContextFormula, Context),
      unify_with_occurs_check(Formula, ContextFormula) ->
        true
    ; LineNumber = 0  % Fallback for missing formulas
    ).
```

5 Rule Implementation Examples

5.1 Modus Ponens (l0mp)

The G4 rule `l0mp` corresponds to a simple modus ponens application:

G4 Structure:

```
l0mp(Sequent, SubProof)
% Where Sequent = (G > [Goal])
% And G contains both (A => B) and A
```

Fitch Translation:

```
render_full_proof_structure(
    l0mp(Sequent, SubProof), Context,
    IndentLevel, CurrentLine, CurrentPos,
    FinalLine, VarIn, VarOut) :-
    Sequent = (G > [Goal]),
    select((A => B), G, G1),
    member(A, G1),
    find_formula_line_safe(Context, (A => B),
                           ImplicationLine),
    find_formula_line_safe(Context, A,
                           AntecedentLine),
    ( ImplicationLine \= 0, AntecedentLine \= 0 ->
        render_have(IndentLevel, B,
                    '$ \\to E $'(ImplicationLine,
                                 AntecedentLine),
                    CurrentLine, NextLine,
                    VarIn, TempVar),
        NewContext = [NextLine:B|Context],
        render_full_proof_structure(SubProof,
                                    NewContext,
                                    IndentLevel, NextLine,
                                    CurrentPos, FinalLine,
                                    TempVar, VarOut)
    ; % Handle error case
        render_full_proof_structure(SubProof, Context,
                                    IndentLevel, CurrentLine,
                                    CurrentPos, FinalLine,
                                    VarIn, VarOut)
    ).
```

Generated Fitch Proof: If we have $A \to B$ at line 1 and A at line 2:

```
A \to B & Given\\
A & Given\\
B & $\to E$ 1, 2\\
```

5.2 Implication Introduction (rcond)

This is more complex as it involves assumption and discharge:

G4 Structure:

```
rcond(Sequent, SubProof)
% Where Sequent = (_ > [(A => B)])
```

Fitch Translation:

```
render_full_proof_structure(
    rcond(Sequent, SubProof), Context,
    IndentLevel, CurrentLine, CurrentPos,
    FinalLine, VarIn, VarOut) :-
    Sequent = (_ > [(A => B)]),
    render_hypo(IndentLevel, A, 'AS',
                CurrentLine, AssumptionLine,
                VarIn, TempVar1),
    NewIndentLevel is IndentLevel + 1,
    NewContext = [AssumptionLine:A|Context],
    render_full_proof_structure(SubProof,
                                NewContext,
                                NewIndentLevel,
                                AssumptionLine,
                                SubCurrentPos,
                                SubProofEnd,
                                TempVar1, TempVar2),
    render_have(IndentLevel, (A => B),
                '$ \\to I $'(AssumptionLine-SubProofEnd),
                SubCurrentPos, FinalLine,
                TempVar2, VarOut),
    CurrentPos = FinalLine.
```

Generated Fitch Proof:

```
\fh A & AS\\
\fa \vdots & \\
\fa B & [SubProof]\\
A \to B & $\to I$ 1-3\\
```

6 Complex Case Study: The ltoto Rule

The `ltoto` rule is the most complex G4 rule, representing the elimination of implications of the form $(A \to B) \to C$.

6.1 Theoretical Background

In G4, the rule is:

$$\frac{A, (B \to C), \Gamma \vdash B \quad C, \Gamma \vdash \Delta}{(A \to B) \to C, \Gamma \vdash \Delta}$$

This corresponds to the classical equivalence:

$$((A \to B) \to C) \equiv (A \to (B \to C)) \land (A \to B)$$

6.2 Fitch Implementation Strategy

The key insight is that to use $(A \to B) \to C$, we must:

1. Construct the implication $A \to B$
2. Apply modus ponens to get C
3. Continue with C available

6.3 Example: The Fundamental Theorem

For the theorem $((A \to B) \to C) \to (B \to C)$:

Input G4 proof:

```
rcond([]>[(((a=>b)=>c)=>b=>c)],
      rcond([((a=>b)=>c)]>[(b=>c)],
            ltoto([b, ((a=>b)=>c)]>[c],
                  ax([a, (b=>c), b]>[b], b),
                  ax([c, b]>[c], c))))
```

Generated Fitch proof:

```
\fh (A \to B) \to C & AS\\
\fa \fh B & AS\\
\fa \fa \fh A & AS\\
\fa \fa A \to B & $\to I$ 3-2\\
\fa \fa C & $\to E$ 1, 4\\
\fa B \to C & $\to I$ 2-5\\
((A \to B) \to C) \to (B \to C) & $\to I$ 1-6\\
```

7 Line Reference Calculation

7.1 The Range Notation

For rules like $\to I$ and $\vee E$, we use range notation:

- `StartLine-EndLine`: Discharge from StartLine to EndLine
- `StartLine1-EndLine1,StartLine2-EndLine2`: Multiple ranges for $\vee E$

7.2 Reference Tracking Algorithm

```
% For implication introduction
render_have(IndentLevel, (A => B),
            '$ \\to I $'(AssumptionLine-ProofEndLine),
            CurrentLine, NextLine, VarIn, VarOut)

% For disjunction elimination
render_have(IndentLevel, Goal,
            '$ \\lor E $'(DisjunctionLine,
                          Case1Start-Case1End,
                          Case2Start-Case2End),
            CurrentLine, NextLine, VarIn, VarOut)
```

8 Error Handling and Robustness

8.1 Missing Formula Handling

The `find_formula_line_safe/3` predicate handles cases where expected formulas are not found:

```
find_formula_line_safe(Context, Formula, LineNum) :-
    ( member(LineNum:Formula, Context) ->
        true
    ; member(LineNum:ContextFormula, Context),
      unify_with_occurs_check(Formula, ContextFormula) ->
        true
    ; LineNum = 0  % Fallback: signal missing formula
    ).
```

When a formula is not found (LineNum = 0), the system can:

1. Skip the rule application
2. Generate an error message
3. Use a fallback strategy

8.2 Context Validation

Before applying any rule, verify that required formulas are available:

```
validate_modus_ponens(Context, A, B) :-
    find_formula_line_safe(Context, (A => B), ImplLine),
    find_formula_line_safe(Context, A, AntLine),
    ImplLine \= 0,
    AntLine \= 0.
```

9 Best Practices

9.1 Variable Naming Convention

Proposed improved naming:

- `IndentLevel` instead of `O`
- `CurrentLine` instead of `N`
- `NextLine` instead of `K`
- `VarCounterIn/VarCounterOut` instead of `U/V`
- `ProofContext` instead of bare `Context`

9.2 Debugging Support

Add debugging predicates:

```
debug_context(Context) :-
    format('Current context:~n'),
    member(Line:Formula, Context),
    format('  Line ~w: ~w~n', [Line, Formula]),
    fail.
debug_context(_).

debug_line_ref(Justification) :-
    format('Generated justification: ~w~n',
           [Justification]).
```

9.3 Incremental Testing

Test each rule implementation independently:

```
test_modus_ponens :-
    Context = [2:a, 1:(a => b)],
    render_full_proof_structure(
        l0mp([a, (a => b)] > [b],
             ax([a, (a => b)] > [b], b)),
        Context, 0, 2, _, FinalLine, 0, _
    ),
    format('Final line: ~w~n', [FinalLine]).
```

10 Advanced Rule Implementations

10.1 Conjunction Elimination (land)

The conjunction elimination rule demonstrates intelligent goal satisfaction:

```
render_full_proof_structure(
    land(Sequent, SubProof), Context,
    IndentLevel, CurrentLine, CurrentPos,
    FinalLine, VarIn, VarOut) :-
    Sequent = (G > [Goal]),
    member((A & B), G),
    find_formula_line_safe(Context, (A & B),
                           ConjunctionLine),

    % Extract both A and B systematically
    render_have(IndentLevel, A,
                '$ \\land E $'(ConjunctionLine),
                CurrentLine, LineA, VarIn, TempVar1),
    render_have(IndentLevel, B,
                '$ \\land E $'(ConjunctionLine),
                LineA, LineB, TempVar1, TempVar2),
    NewContext = [LineA:A, LineB:B|Context],

    % Intelligent goal satisfaction test
    ( goal_satisfied_by_context(Goal, NewContext,
                                DirectLine) ->
        % Goal directly satisfied
        FinalLine = DirectLine,
        CurrentPos = LineB,
        VarOut = TempVar2
    ; goal_satisfiable_in_one_step(Goal, NewContext,
                                   IndentLevel, LineB,
```

```
                                TempVar2, StepLine,
                                StepVar) ->
        % Goal satisfiable in one step
        FinalLine = StepLine,
        CurrentPos = StepLine,
        VarOut = StepVar
    ;
        % Continue with subproof
        render_full_proof_structure(SubProof,
                                    NewContext,
                                    IndentLevel, LineB,
                                    CurrentPos, FinalLine,
                                    TempVar2, VarOut)
    ).
```

10.2 Disjunction Elimination (lor)

This rule requires careful management of multiple assumption contexts:

```
render_full_proof_structure(
    lor(Sequent, SubProof1, SubProof2), Context,
    IndentLevel, CurrentLine, CurrentPos,
    FinalLine, VarIn, VarOut) :-
    Sequent = (G > [Goal]),
    select((A | B), G, G1),
    find_formula_line_safe(Context, (A | B),
                           DisjunctionLine),

    % Case A: Assume A and prove Goal
    render_hypo(IndentLevel, A, 'AS', CurrentLine,
                AssumptionA, VarIn, TempVar1),
    NewIndentLevel is IndentLevel + 1,
    ContextA = [AssumptionA:A|Context],
    render_full_proof_structure(SubProof1, ContextA,
                                NewIndentLevel,
                                AssumptionA, Pos1,
                                ProofEndA, TempVar1,
                                TempVar2),

    % Case B: Assume B and prove Goal
    render_hypo(IndentLevel, B, 'AS', Pos1,
                AssumptionB, TempVar2, TempVar3),
    ContextB = [AssumptionB:B|Context],
    render_full_proof_structure(SubProof2, ContextB,
                                NewIndentLevel,
                                AssumptionB, Pos2,
                                ProofEndB, TempVar3,
                                TempVar4),

    % Apply disjunction elimination
    render_have(IndentLevel, Goal,
                '$ \\lor E $'(DisjunctionLine,
                              AssumptionA-ProofEndA,
                              AssumptionB-ProofEndB),
                Pos2, FinalLine, TempVar4, VarOut),
    CurrentPos = FinalLine.
```

11 Complete System Architecture

11.1 Modular Design

The system is organized into several cooperating modules:

- **fitch$_{\text{printer.pl}}$**: Main proof structure rendering
- **latex$_{\text{utilities.pl}}$**: Formula formatting and LaTeX output
- **g4$_{\text{prover.pl}}$**: G4 proof generation
- **g4$_{\text{printer.pl}}$**: G4 to bussproofs conversion
- **operators.pl**: Operator definitions and precedences

11.2 Data Flow

1. **Input**: Logical formula in TPTP syntax
2. **G4 Proof Generation**: Create sequent calculus proof
3. **Proof Structure Analysis**: Extract rule applications and dependencies
4. **Context Building**: Maintain line-to-formula mappings
5. **Fitch Rendering**: Generate natural deduction proof with correct references
6. **LaTeX Output**: Format for presentation

12 Troubleshooting Common Issues

12.1 Incorrect Line References

Problem: Justifications reference wrong lines or non-existent lines.
Diagnosis:

```
debug_line_reference(Context, Formula, ExpectedLine) :-
    find_formula_line_safe(Context, Formula, ActualLine),
    ( ActualLine = ExpectedLine ->
        format('OK: Formula ~w found at line ~w~n',
               [Formula, ActualLine])
    ; ActualLine = 0 ->
        format('ERROR: Formula ~w not found in context~n',
               [Formula]),
        debug_context(Context)
    ;
        format('WARNING: Formula ~w found at line ~w, expected ~w~n',
               [Formula, ActualLine, ExpectedLine])
    ).
```

Solution: Verify context updates after each rule application.

12.2 Scope Depth Errors

Problem: Too many or too few scope bars in Fitch proof.
Diagnosis: Track scope depth changes:

```
debug_scope_change(OldDepth, NewDepth, Rule) :-
    format('Scope change: ~w -> ~w (Rule: ~w)~n',
           [OldDepth, NewDepth, Rule]).
```

Solution: Ensure scope increases only for assumptions, decreases only for discharges.

12.3 Context Pollution

Problem: Formulas remain in context after their scope ends.

Solution: Implement proper context scoping:

```
scoped_render(Rule, Context, ScopedContext,
              IndentLevel, Result) :-
    % Create new scope
    NewIndentLevel is IndentLevel + 1,
    % Render with scoped context
    render_rule(Rule, ScopedContext, NewIndentLevel, Result),
    % Context automatically reverts when predicate ends
    !.
```

13 Performance Considerations

13.1 Context Search Optimization

For large proofs, context searches can become expensive:

```
% Indexed context for O(log n) lookup
build_indexed_context(Context, IndexedContext) :-
    predsort(compare_by_formula, Context, SortedContext),
    IndexedContext = indexed(SortedContext).

find_formula_indexed(indexed(SortedContext),
                     Formula, Line) :-
    binary_search(SortedContext, Formula, Line).
```

13.2 Memoization

Cache results of expensive operations:

```
:- dynamic formula_line_cache/3.

find_formula_line_memoized(Context, Formula, Line) :-
    ( formula_line_cache(Context, Formula, Line) ->
        true
    ; find_formula_line_safe(Context, Formula, Line),
      assertz(formula_line_cache(Context, Formula, Line))
    ).
```

14 Future Extensions

14.1 Support for Additional Logics

The framework can be extended to support:

- **Modal Logic**: Box and diamond operators
- **Temporal Logic**: Until, always, eventually operators
- **Higher-Order Logic**: Lambda abstractions and applications
- **Intuitionistic Logic**: Restricted classical rules

14.2 Interactive Proof Construction

```
interactive_fitch_builder :-
    read_goal(Goal),
    init_context(Context),
    interactive_step(Goal, Context, 0, 1).

interactive_step(Goal, Context, IndentLevel, CurrentLine) :-
    display_current_state(Goal, Context,
                          IndentLevel, CurrentLine),
    read_user_choice(Choice),
    apply_user_choice(Choice, Goal, Context,
                      IndentLevel, CurrentLine,
                      NewGoal, NewContext,
                      NewIndent, NewLine),
    ( goal_achieved(NewGoal) ->
        display_completed_proof
    ; interactive_step(NewGoal, NewContext,
                       NewIndent, NewLine)
    ).
```

14.3 Proof Verification

Add verification layer:

```
verify_fitch_proof(Proof) :-
    extract_lines(Proof, Lines),
    verify_each_line(Lines, []).

verify_each_line([], _).
verify_each_line([Line|Rest], ValidatedContext) :-
    verify_line(Line, ValidatedContext),
    update_context(Line, ValidatedContext, NewContext),
    verify_each_line(Rest, NewContext).

verify_line(LineNum:Formula:Justification, Context) :-
    verify_justification(Justification, Formula, Context).
```

15 Integration with External Tools

15.1 LaTeX Integration

The system produces publication-ready LaTeX:

```
% Complete document generation
generate_proof_document(Formula, Filename) :-
    prove_formula_clean(Formula, 0, Proof),
    open(Filename, write, Stream),
    write_document_header(Stream),
    current_output(OldOutput),
    set_output(Stream),
    render_lk_to_fitch_direct(Proof, 1),
    write_document_footer(Stream),
    set_output(OldOutput),
    close(Stream).
```

15.2 Web Interface

JSON API for web integration:

```
% REST endpoint for proof generation
handle_prove_request(Request, Response) :-
    json_read_dict(Request, InputDict),
    Formula = InputDict.formula,
    prove_formula_clean(Formula, 0, Proof),
    proof_to_json(Proof, ProofJSON),
    Response = json([status=success, proof=ProofJSON]).
```

16 Conclusion

This comprehensive guide provides the foundation for understanding and extending the automatic Fitch proof construction system. The combination of careful line tracking, robust context management, and systematic rule implementation enables the reliable translation of G4 sequent calculus proofs into readable natural deduction format.

Key success factors include:

1. **Precise variable tracking** with meaningful names
2. **Systematic context management** with proper scoping
3. **Robust error handling** for edge cases
4. **Comprehensive testing** of individual components
5. **Clear documentation** for maintainability

The system serves as both a practical tool for proof presentation and a pedagogical aid for understanding the relationship between different proof systems. Its modular design facilitates future extensions and adaptations to new logical frameworks.

17 Appendices

17.1 Complete Code Templates

17.1.1 Template for New Rule Implementation

```
% Template for implementing a new G4 rule
render_full_proof_structure(
    new_rule(Sequent, SubProofs), Context,
    IndentLevel, CurrentLine, CurrentPos,
    FinalLine, VarIn, VarOut) :-
    !,  % Commit to this clause

    % 1. Extract components from sequent
    Sequent = (Premises > [Goal]),

    % 2. Find required formulas in context
    find_required_formulas(Premises, Context,
                           RequiredLines),

    % 3. Apply rule-specific logic
    apply_rule_logic(RequiredLines, IndentLevel,
                     CurrentLine, IntermediateResults,
                     VarIn, TempVar),

    % 4. Update context with new formulas
    update_context_with_results(IntermediateResults,
```

```
                                Context, NewContext),

    % 5. Process subproofs if needed
    process_subproofs(SubProofs, NewContext,
                      IndentLevel, IntermediateResults,
                      SubResults, TempVar, TempVar2),

    % 6. Generate final result
    generate_final_result(Goal, SubResults,
                          IndentLevel, CurrentPos,
                          FinalLine, TempVar2, VarOut).
```

17.1.2 Context Management Template

```
% Template for context-aware processing
process_with_context(Operation, Context, IndentLevel,
                     CurrentLine, Result, VarIn, VarOut) :-
    % Save current context state
    save_context_state(Context, SavedState),

    % Perform operation with error handling
    catch(
        perform_operation(Operation, Context,
                          IndentLevel, CurrentLine,
                          Result, VarIn, VarOut),
        Error,
        handle_context_error(Error, SavedState,
                             Context, Result,
                             VarIn, VarOut)
    ).

handle_context_error(Error, SavedState, Context,
                     default_result, VarIn, VarIn) :-
    format('Error in context operation: ~w~n', [Error]),
    restore_context_state(SavedState, Context).
```