Simplifying Transformations for Type-alpha Certificates

Konstantine Arkoudas

AI Memo 2001-031 November 2001

Abstract

This paper presents an algorithm for simplifying $\mathcal{NDL}$ deductions. An array of simplifying transformations are rigorously defined. They are shown to be terminating, and to respect the formal semantics of the language. We also show that the transformations never increase the size or complexity of a deduction—in the worst case, they produce deductions of the same size and complexity as the original. We present several examples of proofs containing various types of superfluous "detours", and explain how our procedure eliminates them, resulting in smaller and cleaner deductions. All of the given transformations are fully implemented in SML-NJ. The complete code listing is presented, along with explanatory comments. Finally, although the transformations given here are defined for $\mathcal{NDL}$, we point out that they can be applied to any type-α DPL that satisfies a few simple conditions.

1.1 Introduction

This paper presents a simplification procedure for $\mathcal{NDL}$ [1] deductions. Various transformations are rigorously defined, shown to respect the formal semantics, and fully implemented in SML-NJ. The complete code listing is included, along with explanatory comments. Moreover, although the transformations we present here are defined for $\mathcal{NDL}$, we show how they can be applied to any type-α DPL that satisfies some simple conditions.

Our subject is related to proof-tree normalization in the sense of Prawitz [4] (or alternatively, cut-elimination in sequent-based systems [3, 2]). In the intuitionist case, the Curry-Howard correspondence means that Prawitz normalization coincides with reduction in the simply typed λ-calculus. Accordingly, the normalization algorithm in that case is particularly simple: keep contracting as long as there is a redex (either a β-redex or one of the form $l(\langle e_1, e_2\rangle)$, $r(\langle e_1, e_2\rangle)$, etc.). Strong normalization and the Church-Rosser property guarantee that eventually we will converge to a unique normal form. In the classical case, there is some pre-processing to be done (see Section I, Chapter III of Prawitz's book [4]) before carrying out reductions.

Type-α DPLs such as $\mathcal{NDL}$ present complications of a different nature. One important difference is that in type-α DPLs inference rules are applied to propositions rather than to entire proofs. If $\mathcal{NDL}$ were based on a proof-tree model, where inference rules are applied to proofs, we could then readily formulate local contraction rules in the style of Prawitz, such as

$$\textbf{right-and}(\textbf{both}(D_1, D_2)) \longrightarrow D_2$$
$$\textbf{modus-ponens}(\textbf{assume}\ P \,.\, D_1, D_2) \longrightarrow D_1[D_2/P]$$

and so on. But in $\mathcal{NDL}$ there is not much we can infer from looking at an individual application of an inference rule (such as **left-iff** $P \Leftrightarrow Q$), so global analyses are needed to identify and eliminate detours. Essentially, because assumptions and intermediate conclusions can have limited and arbitrarily nested scopes, it is generally not possible to carry out reductions in a local manner; the overall surrounding context must usually be taken into consideration. Further, the result of one transformation might affect the applicability or outcome of another transformation, so the order in which these occur is important.

Our simplification procedure will consist of a series of transformations, which fall into two groups:

- *restructuring transformations*; and
- *contracting transformations*, or simply *contractions.*

Contracting transformations form the bedrock of the simplification process: they remove extraneous parts, thereby reducing the size and complexity of a deduction. Restructuring transformations simply rearrange the structure of a deduction so as to better expose simplification opportunities; they constitute a kind of pre-processing aimed at facilitating the contracting transformations.

Specifically, our top-level simplification procedure is defined as follows:

$$\mathit{simplify} = \mathit{contract} \cdot \mathit{restructure} \tag{1.1}$$

where $\cdot$ denotes ordinary function composition and

$$\begin{aligned} \mathit{contract} &= \mathit{fp}\ \ (\mathfrak{C} \cdot \mathfrak{P} \cdot \mathfrak{U}) && (1.2)\\ \mathit{restructure} &= \mathit{reduce}\ \ (\lambda\, f, g \,.\, \mathfrak{MS} \cdot f \cdot g)\ \ \mathfrak{MS}\ \ [\mathfrak{A}_3, \mathfrak{A}_2, \mathfrak{A}_1]. && (1.3)\end{aligned}$$

The fixed-point-finder function *fp* is defined as:

```
$fp\ \ f = \lambda\, D \,.\, let\ \ D' = f\ \ D$
            $in$
              $D = D'\ ? \rightarrow D,\ fp\ \ f\ \ D'$
```

while *reduce* is the usual list-reducing functional. An equivalent definition of *restructure* is as follows:

$$\mathit{restructure} = \mathit{weave}\ \ \mathfrak{MS}\ \ [\mathfrak{A}_3, \mathfrak{A}_2, \mathfrak{A}_1] \tag{1.4}$$

with the weaving function defined thus:

```
$weave\ \ f\ \ L = let\ \ T\ \ [] = f$
                    $T\ \ g{::}L' = f \cdot g \cdot (T\ \ L')$
              $in$
                $T\ \ L$
```

We will continue to define functions in this informal notation, using pattern matching, recursion, etc., in the style of (strict) higher-order functional languages such as ML. Any reader moderately familiar with a programming language of this kind should be able to make sense of our definitions. Complete SML-NJ code will be given in Section 1.6. As a convention, we will write $E\,? \rightarrow E_1,\ E_2$ to mean "if E then E_1 else E_2". Also, we write $[x_1, \ldots, x_n]$ for the list of any $n \geq 0$ elements $x_1, \ldots, x_n$, and $x{::}L$ for the list obtained by prepending ("consing") x in front of L. Finally, we will use the symbol $\oplus$ to denote list concatenation.

We will show that our simplification procedure has three important properties: it always terminates; it is safe; and it never increases the size or complexity of a deduction. Specifically, the following will hold for all deductions D:

1. The computation of *simplify*(D) terminates.

2. *simplify*(D) respects the semantics of D, in a sense that will be made rigorous in Section 1.2.

3. The size of *simplify*(D) is less than or equal to the size of D.

The remainder of this paper is structured as follows. The next section briefly reviews the syntax and semantics of $\mathcal{NDL}$, along with some basic notions and results that will form the theoretical background for our transformations. The following two sections discuss each group of transformations in turn: first the contractions $\mathfrak{C}$, $\mathfrak{P}$, and $\mathfrak{U}$; and then the restructuring transformations $\mathfrak{MS}$, $\mathfrak{A}_1$, $\mathfrak{A}_2$,

$$\begin{array}{lcl}
\textit{Prim-Rule} & ::= & \textbf{claim} \mid \textbf{modus-ponens} \mid \textbf{cond} \mid \textbf{neg} \mid \textbf{true-intro} \\
 & \mid & \textbf{both} \mid \textbf{left-and} \mid \textbf{right-and} \mid \textbf{double-negation} \\
 & \mid & \textbf{cd} \mid \textbf{left-either} \mid \textbf{right-either} \mid \textbf{equivalence} \\
 & \mid & \textbf{left-iff} \mid \textbf{right-iff} \mid \textbf{absurd}
\end{array}$$

Figure 1.1: The primitive inference rules of $\mathcal{NDL}$.

and $\mathfrak{A}_3$. In Section 1.5 we give a number of examples illustrating the various transformations in action; the examples demonstrate that *simplify* can often result in dramatic size reductions. Finally, in Section 1.6 we present SML-NJ code that implements all of the given transformations, and we discuss how these ideas can be applied to other type-α DPLs. An appendix offers rigorous definitions of some $\mathcal{NDL}$ concepts that appear informally in Section 1.2.

1.2 Review of $\mathcal{NDL}$

Propositions are defined by the following abstract grammar:

$$P ::= A \mid \textbf{true} \mid \textbf{false} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid P \Leftrightarrow Q$$

where A ranges over some unspecified, countable set of atomic propositions ("atoms"). The letters A, B, and C will be used as typical atoms, and P, Q, and R as propositions. Parsing ambiguities will be resolved by parentheses and brackets. By an *assumption base* β we will mean a finite set of propositions.

The proofs (or "deductions") of $\mathcal{NDL}$ have the following abstract syntax:

$$\begin{array}{rcll}
D & = & \textit{Prim-Rule}\ P_1, \ldots, P_n & \text{(Primitive rule applications)} \\
 & \mid & \textbf{assume}\ P\ \textbf{in}\ D & \text{(Conditional deductions)} \\
 & \mid & \textbf{suppose-absurd}\ P\ \textbf{in}\ D & \text{(Proofs by contradiction)} \\
 & \mid & D_1; D_2 & \text{(Compositions)}
\end{array}$$

where *Prim-Rule* ranges over the collection of primitive inference rules shown in Figure 1.1.[1] Deductions of the form *Prim-Rule* $P_1, \ldots, P_n$ are called *primitive rule applications*; those of the form **assume** P **in** D and $D_1; D_2$ are *hypothetical* and *composite* deductions, respectively; and those of the form **suppose-absurd** P **in** D are called deductions *by contradiction*. Primitive rule applications are *atomic* deductions, as they have no recursive structure, whereas all other forms are *compound* or *complex*. Conditional deductions of the form **assume** P **in** D and proofs by contradiction of the form **suppose-absurd** P **in** D will be called *hypothetical deductions*. In both cases, P and D are called the *hypothesis* and *body* of the deduction, respectively; we also say that the body D represents the *scope* of the hypothesis P. A *trivial* deduction is a claim, i.e., an atomic deduction of the form **claim** P. A deduction is *well-formed* iff every primitive rule application in it has one of the forms shown in Figure 1.12 (a precise definition appears in the Appendix). It is straightforward to check whether a deduction is well-formed, and from now on we will assume that all deductions are well-formed. We stipulate that the composition operator is right-associative. A maximal-length

[1] Note that **modus-tollens** and **false-elim** are not needed here, since we are taking **suppose-absurd** as a primitive. Also, **cond** and **neg**, which are used to introduce conditionals and negations, respectively, are not found in other presentations of $\mathcal{NDL}$. Consult the Appendix for a discussion of these two rules.

$$
\begin{aligned}
FA(D_1; D_2) &= FA(D_1) \cup (FA(D_2) - \{\mathcal{C}(D_1)\}) && (1.5)\\
FA(\textbf{assume } P \textbf{ in } D) &= FA(D) - \{P\} && (1.6)\\
FA(\textbf{suppose-absurd } P \textbf{ in } D) &= FA(D) - \{P\} && (1.7)\\
FA(\textbf{left-either } P_1, P_2) &= \{P_1\} && (1.8)\\
FA(\textbf{right-either } P_1, P_2) &= \{P_2\} && (1.9)\\
FA(\textit{Prim-Rule } P_1, \ldots, P_n) &= \{P_1, \ldots, P_n\} && (1.10)
\end{aligned}
$$

Figure 1.2: Definition of $FA(D)$ for $\mathcal{NDL}$ ($\textit{Prim-Rule} \notin \{\textbf{left-either}, \textbf{right-either}\}$ in 1.10).

composition $D_1; \ldots; D_n$ is called a *thread*. The last element of a thread is said to be in a *tail position*. Ambiguities in the parsing of $\mathcal{NDL}$ deductions will be resolved by the use of **begin-end** pairs.

The semantics of $\mathcal{NDL}$ are given by judgments of the form $\beta \vdash D \leadsto P$, which are understood to say "Evaluating D in β produces the conclusion P." The semantics of rule applications appear in Figure 1.12, in the Appendix. The semantics of compound deductions are given by the following three rules:

$$
\frac{\beta \cup \{P\} \vdash D \leadsto Q}{\beta \vdash \textbf{assume } P \textbf{ in } D \leadsto P \Rightarrow Q}\ [R_1] \qquad \frac{\beta \cup \{P\} \vdash D \leadsto \textbf{false}}{\beta \vdash \textbf{suppose-absurd } P \textbf{ in } D \leadsto \neg P}\ [R_2]
$$

$$
\frac{\beta \vdash D_1 \leadsto P_1 \qquad \beta \cup \{P_1\} \vdash D_2 \leadsto P_2}{\beta \vdash D_1; D_2 \leadsto P_2}\ [R_3]
$$

We have:

Theorem 1.1 (Dilution) *If $\beta \vdash D \leadsto P$ then $\beta \cup \beta' \vdash D \leadsto P$.*

The *conclusion* of a deduction D, denoted $\mathcal{C}(D)$, is defined by structural recursion. For compound deductions we have:

$$
\begin{aligned}
\mathcal{C}(\textbf{assume } P \textbf{ in } D) &= P \Rightarrow \mathcal{C}(D)\\
\mathcal{C}(\textbf{suppose-absurd } P \textbf{ in } D) &= \neg P\\
\mathcal{C}(D_1; D_2) &= \mathcal{C}(D_2)
\end{aligned}
$$

while rule applications are covered in the Appendix.

Theorem 1.2 *If $\beta \vdash D \leadsto P$ then $P = \mathcal{C}(D)$.*

Proof: A straightforward induction on D. ■

Corollary 1.3 *If $\beta \vdash D \leadsto P_1$ and $\beta \vdash D \leadsto P_2$ then $P_1 = P_2$.*

Figure 1.2 defines $FA(D)$, the set of *free assumptions* of a proof D. The elements of $FA(D)$ are propositions that D uses as premises, without proof. Note in particular equations 1.6 and 1.7, for hypothetical deductions: the free assumptions here are those of the body D *minus* the hypothesis P. We will say that the elements of $FA(D)$ are *strictly used* by D.

Theorem 1.4 *$\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ iff $FA(D) \subseteq \beta$.*

Proof: By induction on the structure of D. ■

We say that two deductions D_1 and D_2 are observationally equivalent with respect to an assumption base β, written $D_1 \approx_\beta D_2$, whenever

$$\beta \vdash D_1 \rightsquigarrow P \quad \text{iff} \quad \beta \vdash D_2 \rightsquigarrow P$$

for all P. We say that D_1 and D_2 are *observationally equivalent*, written $D_1 \approx D_2$, iff we have $D_1 \approx_\beta D_2$ for all β.

Lemma 1.5 *If $D_1 \approx D_2$ then $\mathcal{C}(D_1) = \mathcal{C}(D_2)$.*

Proof: Set $\beta = FA(D_1) \cup FA(D_2)$. By Theorem 1.4, we have $\beta \vdash D_1 \rightsquigarrow \mathcal{C}(D_1)$, so the assumption $D_1 \approx D_2$ entails $\beta \vdash D_2 \rightsquigarrow \mathcal{C}(D_1)$. But Theorem 1.4 also gives $\beta \vdash D_2 \rightsquigarrow \mathcal{C}(D_2)$, hence $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ by Corollary 1.3. ■

Theorem 1.6 *$D_1 \approx D_2$ iff $FA(D_1) = FA(D_2)$ and $\mathcal{C}(D_1) = \mathcal{C}(D_2)$. Therefore, observational equivalence is decidable.*

Proof: In one direction, suppose that $FA(D_1) = FA(D_2)$ and $\mathcal{C}(D_1) = \mathcal{C}(D_2)$. Then, for any β and P, we have:

$$\begin{array}{lll}
\beta \vdash D_1 \rightsquigarrow P & \text{iff} & \text{(by Theorem 1.2 and Theorem 1.4)} \\
P = \mathcal{C}(D_1) \text{ and } \beta \supseteq FA(D_1) & \text{iff} & \text{(by the assumptions } \mathcal{C}(D_1) = \mathcal{C}(D_2) \text{ and } FA(D_1) = FA(D_2)) \\
P = \mathcal{C}(D_2) \text{ and } \beta \supseteq FA(D_2) & \text{iff} & \text{(by Theorem 1.2 and Theorem 1.4)} \\
\beta \vdash D_2 \rightsquigarrow P. & &
\end{array}$$

This shows that $D_1 \approx D_2$.

Conversely, suppose that $D_1 \approx D_2$. Then $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ follows from Lemma 1.5. Moreover, by Theorem 1.4, $FA(D_1) \vdash D_1 \rightsquigarrow \mathcal{C}(D_1)$, so the assumption $D_1 \approx D_2$ entails $FA(D_1) \vdash D_2 \rightsquigarrow \mathcal{C}(D_1)$. Therefore, by Theorem 1.4,

$$FA(D_1) \supseteq FA(D_2). \tag{1.11}$$

Likewise, we have $FA(D_2) \vdash D_2 \rightsquigarrow \mathcal{C}(D_2)$, so $D_1 \approx D_2$ implies $FA(D_2) \vdash D_1 \rightsquigarrow \mathcal{C}(D_2)$, and hence Theorem 1.4 gives

$$FA(D_2) \supseteq FA(D_1) \tag{1.12}$$

and now $FA(D_1) = FA(D_2)$ follows from 1.11 and 1.12. ■

Observational equivalence is a very strong condition. Oftentimes we are only interested in replacing a deduction D_1 by some D_2 on the assumption that D_1 will yield its conclusion in the intended β (i.e., on the assumption that its evaluation will not lead to error), even though we might have $D_1 \not\approx D_2$. To take a simple example, although we have **claim** $P; D \not\approx D$ (pick D to be **true-intro** and consider any β that does not contain P), it is true that in any given assumption base, *if* **claim** $P; D \not\approx D$ produces some conclusion Q *then* so will D. (In fact this observation will be the formal justification for a transformation we will introduce later for removing redundant claims.) We formalize this relation as follows.

We write $D_1 \rightarrowtail_\beta D_2$ to mean that, for all P,

$$\text{if } \beta \vdash D_1 \leadsto P \text{ then } \beta \vdash D_2 \leadsto P.$$

That is, $D_1 \rightarrowtail_\beta D_2$ holds if $\mathit{Eval}(D_1, \beta) = \mathit{Eval}(D_2, \beta)$ whenever $\mathit{Eval}(D_1, \beta) = \mathcal{C}(D_1)$, or, equivalently, whenever $\mathit{Eval}(D_1, \beta) \neq \mathit{error}$. And we will write $D_1 \rightarrowtail D_2$ to mean that $D_1 \rightarrowtail_\beta D_2$ for *all* β.

Clearly, $\rightarrowtail$ is not a symmetric relation: we vacuously have **claim** $A \rightarrowtail$ **true-intro**, but the converse does not hold. However, $\rightarrowtail$ is a quasi-order (reflexive and transitive), and in fact $\approx$ is the contensive equality generated by the weaker relation's symmetric closure.

Lemma 1.7 *$\rightarrowtail$ is a quasi-order whose symmetric closure coincides with $\approx$. Accordingly, $D_1 \approx D_2$ iff $D_1 \rightarrowtail D_2$ and $D_2 \rightarrowtail D_1$.*

It will be useful to note that $\rightarrowtail$ is compatible with the syntactic constructs of $\mathcal{NDL}$:

Lemma 1.8 *If $D_1 \rightarrowtail D_1'$, $D_2 \rightarrowtail D_2'$ then* **assume** P **in** $D_1 \rightarrowtail$ **assume** P **in** D_1'*, $D_1; D_2 \rightarrowtail D_1'; D_2'$, and* **suppose-absurd** P **in** $D_1 \rightarrowtail$ **suppose-absurd** P **in** D_1'*.*

Reasoning similar to that used in the proof of Theorem 1.4 will show:

Theorem 1.9 *$D_1 \rightarrowtail D_2$ iff $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ and $FA(D_1) \supseteq FA(D_2)$. Therefore, the relation $\rightarrowtail$ is decidable.*

Finally, the following result will help us to justify a "hoisting" transformation that we will define later:

Theorem 1.10 *If $P \notin FA(D_1)$ then* (a) **assume** P **in** $(D_1; D_2) \rightarrowtail D_1;$ **assume** P **in** D_2*, and*

(b) **suppose-absurd** P **in** $(D_1; D_2) \rightarrowtail D_1;$ **suppose-absurd** P **in** D_2.

Proof: We prove part (a); part (b) is similar. Suppose $\beta \vdash$ **assume** P **in** $(D_1; D_2) \leadsto P \Rightarrow Q$, so that $\beta \cup \{P\} \vdash D_1; D_2 \leadsto Q$. Accordingly,

$$\beta \cup \{P\} \vdash D_1 \leadsto P_1 \tag{1.13}$$

and

$$\beta \cup \{P\} \cup \{P_1\} \vdash D_2 \leadsto Q \tag{1.14}$$

(where, of course, $P_1 = \mathcal{C}(D_1)$, $Q = \mathcal{C}(D_2)$). Thus 1.14 gives

$$\beta \cup \{P_1\} \vdash \textbf{assume } P \textbf{ in } D_2 \leadsto P \Rightarrow Q. \tag{1.15}$$

From 1.13 and Theorem 1.4, $\beta \cup \{P\} \supseteq FA(D_1)$, hence, since $P \notin FA(D_1)$,

$$\beta \supseteq FA(D_1). \tag{1.16}$$

Therefore,

$$\beta \vdash D_1 \leadsto P_1 \tag{1.17}$$

so, from 1.15 and 1.17, rule $[R_3]$ gives $\beta \vdash D_1;$ **assume** P **in** $D_2 \leadsto P \Rightarrow Q$, which establishes (a). ■

The relation $\rightarrowtail$ will serve as our formal notion of safety for the transformations that will be introduced. That is, whenever a transformation maps a deduction D_1 to some D_2, we will have $D_1 \rightarrowtail D_2$. This is an appropriate notion of safety in the context of certificates, because if D_1 is a certificate then presumably we already know that it works; we are only interested in making it more efficient or succinct. For other applications, however, if we wish our transformations to be perfectly safe then we should insist on observational equivalence. For $D_1 \approx D_2$ means that the two deductions behave identically in all contexts, i.e., in all assumption bases. For any β, if D_1 fails in β then D_2 will fail in β as well; while if D_1 produces a conclusion P in β, then D_2 will produce that same conclusion in β. Accordingly, the replacement of D_1 by D_2 would be a completely semantics-preserving transformation.

1.3 Contracting transformations

Informally, our contracting transformations will be based on two simple principles:

Productivity: *Every intermediate conclusion should be used at some later point as an argument to a primitive inference rule.*

Parsimony: *At no point should a non-trivial deduction establish something that has already been established, or something that has been hypothetically postulated.*

These principles are respectively based on the notions of *redundancies* and *repetitions*, which we will now study in detail.

Redundancies

Intuitively, a deduction contains redundancies if it generates conclusions which are not subsequently used. For all practical purposes, such conclusions are useless "noise". We will see that they can be systematically eliminated. Redundancy-free deductions will be called *strict*. As a very simple example, the following deduction, which proves $A \wedge B \Rightarrow A$, is not strict:

$$\textbf{assume } A \wedge B \textbf{ in begin right-and } A \wedge B; \textbf{left-and } A \wedge B; \textbf{end}$$

The redundancy here is the application of **right-and** to derive B. This is superfluous because it plays no role in the derivation of the final conclusion. We formally define the judgment $\vdash_S D$, "D is strict", in Figure 1.3. Verbally, the definition can be phrased as follows:

- *Atomic deductions are always strict.*
- *A hypothetical deduction is strict if its body is strict.*
- *A composite deduction $D_1; D_2$ is strict if both D_1 and D_2 are strict, and the conclusion of D_1 is strictly used in D_2.*

The last of the above clauses is the most important one. Note that we require that $\mathcal{C}(D_1)$ be *strictly* used in D_2. Accordingly, the deduction

$$\textbf{left-and } A \wedge B; \textbf{ assume } A \textbf{ in both } A, A$$

is not strict: the derivation of A via **left-and** is extraneous because the only subsequent use of A, as a premise to **both** inside the **assume**, has been "buffered" by the hypothetical postulation of A.

$$\frac{}{\vdash_S \mathit{Prim\text{-}Rule}\ P_1, \ldots, P_n} \qquad \frac{\vdash_S D}{\vdash_S \textbf{assume } P \textbf{ in } D}$$

$$\frac{\vdash_S D}{\vdash_S \textbf{suppose-absurd } P \textbf{ in } D} \qquad \frac{\vdash_S D_1 \quad \vdash_S D_2 \quad \mathcal{C}(D_1) \in \mathit{FA}(D_2)}{\vdash_S D_1; D_2}$$

Figure 1.3: Definition of *strict* deductions.

We will now present a transformation algorithm $\mathfrak{U}$ that converts a given deduction D into a strict deduction D'. We will prove that $\vdash_S D'$, and also that the semantics of D are conservatively preserved in the sense that $D \rightarrowtail D'$. The transformation is defined by structural recursion:

$$\begin{aligned}
&\mathfrak{U}(\textbf{assume } P \textbf{ in } D) = \textbf{assume } P \textbf{ in } \mathfrak{U}(D)\\
&\mathfrak{U}(\textbf{suppose-absurd } P \textbf{ in } D) = \textbf{suppose-absurd } P \textbf{ in } \mathfrak{U}(D)\\
&\mathfrak{U}(D_1; D_2) = \mathit{let}\ \ D_1' = \mathfrak{U}(D_1)\\
&\qquad\qquad\qquad\quad\ \ D_2' = \mathfrak{U}(D_2)\\
&\qquad\qquad\quad\ \ \mathit{in}\\
&\qquad\qquad\qquad \mathcal{C}(D_1') \notin \mathit{FA}(D_2') \to D_2', D_1'; D_2'\\
&\mathfrak{U}(D) = D
\end{aligned}$$

Informally, it is easy to see that $D \rightarrowtail \mathfrak{U}(D)$ because $\mathfrak{U}(D)$ does not introduce any additional free assumptions (though it might eliminate some of the free assumptions of D), and does not alter $\mathcal{C}(D)$. Therefore, by Theorem 1.9, we have $D \rightarrowtail \mathfrak{U}(D)$. More precisely:

Theorem 1.11 (a) $\mathfrak{U}$ *always terminates;* (b) $\mathfrak{U}(D)$ *is strict;* (c) $D \rightarrowtail \mathfrak{U}(D)$.

Proof: Termination is clear, since the size of the argument strictly decreases with each recursive call. We prove (b) and (c) simultaneously by structural induction on D.

The base case of atomic deductions is immediate. When D is of the form **assume** P **in** D_b, we have

$$\mathfrak{U}(D) = \textbf{assume } P \textbf{ in } \mathfrak{U}(D_b). \tag{1.18}$$

By the inductive hypothesis, $\mathfrak{U}(D_b)$ is strict, hence so is $\mathfrak{U}(D)$, by the definition of strictness. Further, again by the inductive hypothesis, we have $D_b \rightarrowtail \mathfrak{U}(D_b)$, hence by Lemma 1.8 we get

$$\textbf{assume } P \textbf{ in } D_b \rightarrowtail \textbf{assume } P \textbf{ in } \mathfrak{U}(D_b)$$

which is to say, by virtue of 1.18, that $D \rightarrowtail \mathfrak{U}(D)$. The reasoning for proofs by contradiction is similar.

Finally, suppose that D is a composite deduction $D_1; D_2$ and let $D_1' = \mathfrak{U}(D_1), D_2' = \mathfrak{U}(D_2)$. Either $\mathcal{C}(D_1') \in \mathit{FA}(D_2')$ or not. If so, then $\mathfrak{U}(D) = D_1'; D_2'$, and strictness follows from the inductive hypothesis and our supposition that $\mathcal{C}(D_1') \in \mathit{FA}(D_2')$, according to the definition of $\vdash_S$; while $D \rightarrowtail \mathfrak{U}(D)$ in this case means $D_1; D_2 \rightarrowtail D_1'; D_2'$, which follows from the inductive hypotheses in tandem with Lemma 1.8. By contrast, suppose that $\mathcal{C}(D_1') \notin \mathit{FA}(D_2')$, so that $\mathfrak{U}(D) = D_2'$. Since $D = D_1; D_2 \rightarrowtail D_1'; D_2'$ follows from the inductive hypotheses and Lemma 1.8, if we can show that $D_1'; D_2' \rightarrowtail D_2'$ then $D \rightarrowtail D_2' = \mathfrak{U}(D)$ will follow from the transitivity of $\rightarrowtail$ (Lemma 1.7). Accordingly, pick any β and Q, and suppose that $\beta \vdash D_1'; D_2' \rightsquigarrow Q$ (where, of course, by Theorem 1.2 we must have $Q = \mathcal{C}(D_1'; D_2') = \mathcal{C}(D_2')$). By Theorem 1.4, this means that

$$\beta \supseteq FA(D_1'; D_2'). \tag{1.19}$$

But the supposition $\mathcal{C}(D_1') \notin FA(D_2')$ entails, by the definition of free assumptions, that $FA(D_1'; D_2') = FA(D_1') \cup FA(D_2')$, so 1.19 gives $\beta \supseteq FA(D_2')$. Therefore, Theorem 1.4 implies $\beta \vdash D_2' \rightsquigarrow \mathcal{C}(D_2') = Q$. We have thus shown that for any β and Q, if $\beta \vdash D_1'; D_2' \rightsquigarrow Q$ then $\beta \vdash D_2' \rightsquigarrow Q$, which is to say $D_1'; D_2' \rightarrowtail D_2'$. It follows from our earlier remarks that $D = D_1; D_2 \rightarrowtail D_2' = \mathfrak{U}(D)$. This completes the inductive argument. ■

As an illustration, suppose we wish to use the algorithm to remove redundancies from the deduction

$$D_1; D_2; \textbf{both } A, B; \textbf{left-either } A, C \tag{1.20}$$

where $\mathcal{C}(D_1) = A, \mathcal{C}(D_2) = B$. Assuming that D_1 and D_2 are already strict, the interesting reduction steps taken by the algorithm, in temporal order, may be depicted as follows (where we use the arrow $\Longrightarrow$ to represent a reduction step):

1. $\textbf{both } A, B; \textbf{left-either } A, C \Longrightarrow \textbf{left-either } A, C$ (as $A \wedge B \notin FA(\textbf{left-either } \text{A, C}))$
2. $D_2; \textbf{left-either } A, C \Longrightarrow \textbf{left-either } A, C$ (as $\mathcal{C}(D_2) = B \notin FA(\textbf{left-either } \text{A, C}))$
3. $D_2; \textbf{both } A, B; \textbf{left-either } A, C \Longrightarrow D_2; \textbf{left-either } A, C$ (from 1)
4. $D_2; \textbf{both } A, B; \textbf{left-either } A, C \Longrightarrow \textbf{left-either } A, C$ (from 2 and 3)
5. $D_1; D_2; \textbf{both } A, B; \textbf{left-either } A, C \Longrightarrow D_1; \textbf{left-either } A, C$ (from 4)

Thus the original deduction becomes reduced to $D_1; \textbf{left-either } A, C$.

Repetitions

The principle of productivity alone cannot guarantee that a deduction will not have superfluous components. For instance, consider a slight modification of example 1.20:

$$D_1; D_2; \textbf{both } A, B; \textbf{left-and } A \wedge B \tag{1.21}$$

where again $\mathcal{C}(D_1) = A$, $\mathcal{C}(D_2) = B$. The difference with 1.20 is that the last deduction is

$$\textbf{left-and } A \wedge B$$

instead of **left-either** A, C. In this case algorithm $\mathfrak{U}$ will have no effect because the deduction is already strict: D_1 establishes A; D_2 establishes B; then we use both A and B to obtain $A \wedge B$; and finally we use **left-and** $A \wedge B$ to get A. Thus the principle of productivity is observed. The principle of parsimony, however, is clearly violated: the **left-and** deduction establishes something (A) which has already been established by D_1. For that reason, it is superfluous, and hence so are the derivations of B and $A \wedge B$.

This example illustrates what Prawitz called a *detour*: the gratuitous application of an introduction rule followed by the application of a corresponding elimination rule that gets us back to a premise which we had supplied to the introduction rule. The reason why these are detours is because elimination rules are the inverses of introduction rules. Prawitz enunciated this intuition with an informal statement

```
$D_1$;   (proves $P$)
⋮
suppose-absurd $\neg P$ in
  $D_2$;
⋮
double-negation $\neg\neg P$;
⋮
```
(a) Detour for $\neg$.

```
$D_1$;   (proves $P$)
⋮
$D_2$;   (proves $Q$)
⋮
both $P, Q$;
⋮
left-and $P \wedge Q$;
⋮
```
(b) Detour for $\wedge$.

```
$\{D_1$;   (proves $Q\}$)
⋮
$D_2$;   (proves $P$)
⋮
assume $P$ in
  $D_3$; (proves $Q$)
⋮
modus-ponens $P \Rightarrow Q, P$;
⋮
```
(c) Detour for $\Rightarrow$.

```
$D_1$;   (proves $P$)
⋮
left-either $P, Q$;
⋮
assume $P$ in
  $D_2$;   (proves $Q'$)
⋮
assume $Q$ in
  $D_3$;   (proves $Q'$)
⋮
cd $P \vee Q, P \Rightarrow Q', Q \Rightarrow Q'$;
⋮
```
(d) Detour for $\vee$.

```
assume $P$ in
  $D_1$;   (proves $Q$)
⋮
assume $Q$ in
  $D_2$;   (proves $P$)
⋮
equivalence $P \Rightarrow Q, Q \Rightarrow P$;
⋮
left-iff $P \Leftrightarrow Q$;
⋮
```
(e) Detour for $\Leftrightarrow$.

Figure 1.4: Prawitz-type detours for $\mathcal{NDL}$.

that he called "the inversion principle". Figure 1.4 shows the form that Prawitz's detours take in $\mathcal{NDL}$ for each of the five connectives. For $\wedge$, $\vee$, and $\Leftrightarrow$ there are twin detours, which we do not depict here, where **right-and**, **right-either**, and **right-iff** take the place of **left-and**, **left-either**, and **left-iff**, respectively. Furthermore, the detour contained in each of the threads shown in Figure 1.4 is insensitive to the ordering of most of the thread's elements: for instance, in thread (b) we may be able to swap D_1 and D_2, but this would not affect the detour; in (c) we might put D_1 immediately before the **modus-ponens**, but we would still have the same detour; and so on. So the threads of Figure 1.4 should be understood up to some permutation of the depicted elements (of course one ordering constraint that must always be respected is that elimination rules should come after introduction rules). Finally, D_1 in (c) is optional, indicated by the braces around it; we would still have essentially the same detour in the absence of D_1.

It is important to realize that Prawitz's reductions are not readily applicable in $\mathcal{NDL}$. Detours may not be freely replaced by their obvious contractions; the greater context in which the subdeduction occurs will determine whether the replacement is permissible. For example, the boxed subdeduction below indicates a detour, but we may not blindly simplify it because $\mathcal{C}(D_2)$, or $\mathcal{C}(D_1) \wedge \mathcal{C}(D_2)$, or

```
$\mathfrak{P}(D) = RR(D, \emptyset)$
where
$RR(D, \Phi) = \mathcal{C}(D) \in \Phi \rightarrow$ claim $\mathcal{C}(D)$,
                     match $D$
                         assume $P$ in $D_b \rightarrow$ assume $P$ in $RR(D_b, \Phi \cup \{P\})$
                         suppose-absurd $P$ in $D_b \rightarrow$ suppose-absurd $P$ in $RR(D_b, \Phi \cup \{P\})$
                         $D_1; D_2 \rightarrow$ let $D_1' = RR(D_1, \Phi)$
                                  in
                                    $D_1'; RR(D_2, \Phi \cup \{\mathcal{C}(D_1')\})$
                         $D \rightarrow D$
```

Figure 1.5: Algorithm for removing redundancies.

both, might be needed inside D':

$$\cdots; D_1; \boxed{D_2; \textbf{both } \mathcal{C}(D_1), \mathcal{C}(D_2); \textbf{left-and } \mathcal{C}(D_1) \wedge \mathcal{C}(D_2)}; \cdots D' \cdots$$

What we *can* do, however, is replace the inference **left-and** $\mathcal{C}(D_1) \wedge \mathcal{C}(D_2)$ by the trivial **claim** $\mathcal{C}(D_1)$. A subsequent strictness analysis will determine whether $\mathcal{C}(D_2)$ or $\mathcal{C}(D_1) \wedge \mathcal{C}(D_2)$ are needed at any later point. If not, then we can be sure that the deductions D_2 and **both** $\mathcal{C}(D_1), \mathcal{C}(D_2)$ were indeed a detour, and algorithm $\mathfrak{U}$ will eliminate them. We will see that this simple technique of

1. replacing every deduction whose conclusion P has already been established by the trivial deduction that claims P, and then

2. removing redundancies with our productivity analysis

will be sufficient for the elimination of most of the detours shown in Figure 1.4. The first step can result in a deduction with various trivial claims sprinkled throughout. This is mostly a cosmetic annoyance; a simple contracting analysis that we will present shortly will eliminate all extraneous claims. That analysis will always be performed at the end of all other transformations in order to clean up the final result.

We now present an algorithm $\mathfrak{P}$, in Figure 1.5, for performing the first step of the above process. The following lemma will be useful in proving the correctness of this transformation.

Lemma 1.12 *If* $\beta \vdash RR(D, \Phi) \rightsquigarrow Q$ *then* $\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \rightsquigarrow Q$.

Proof: By structural induction on D. When D is an atomic deduction, we distinguish two cases: either $\mathcal{C}(D) \in \Phi$ or not. In the former case we have $RR(D, \Phi) =$ **claim** $\mathcal{C}(D)$, so the assumption $\beta \vdash RR(D, \Phi) \rightsquigarrow Q = \mathcal{C}(D)$ means that

$$\beta \vdash \textbf{claim } \mathcal{C}(D) \rightsquigarrow \mathcal{C}(D). \tag{1.22}$$

Now since we are assuming $\mathcal{C}(D) \in \Phi$, we have $\mathcal{C}(D) \in \Phi \cup \{P\}$, hence

$$RR(D, \Phi \cup \{P\}) = \textbf{claim } \mathcal{C}(D)$$

so we need to show that $\beta \cup \{P\} \vdash \textbf{claim } \mathcal{C}(D) \rightsquigarrow \mathcal{C}(D)$. But this follows from 1.22 by dilution. By contrast, if $\mathcal{C}(D) \notin \Phi$ then $RR(D, \Phi) = D$, so we are assuming that

$$\beta \vdash D \rightsquigarrow \mathcal{C}(D) \tag{1.23}$$

and we need to show that

$$\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \rightsquigarrow \mathcal{C}(D). \tag{1.24}$$

Now there are two subcases, namely, either $\mathcal{C}(D) = P$ or not. If $\mathcal{C}(D) = P$ then

$$RR(D, \Phi \cup \{P\}) = \textbf{claim } P$$

and 1.24 follows from the semantics of **claim**. If $\mathcal{C}(D) \neq P$ then $RR(D, \Phi \cup \{P\}) = D$ (since then $\mathcal{C}(D) \notin \Phi \cup \{P\}$), so 1.24 is tantamount to $\beta \cup \{P\} \vdash D \rightsquigarrow \mathcal{C}(D)$, which follows from 1.23 by dilution. This completes both case analyses.

When D is of the form **assume** P_1 **in** D_1, the assumption $\beta \vdash RR(D, \Phi) \rightsquigarrow Q$ translates to

$$\beta \vdash \textbf{assume } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P_1\}) \rightsquigarrow P_1 \Rightarrow P_2 \tag{1.25}$$

where

$$\beta \cup \{P_1\} \vdash RR(D_1, \Phi \cup \{P_1\}) \rightsquigarrow P_2. \tag{1.26}$$

Now $RR(D, \Phi \cup \{P\}) = \textbf{assume } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P, P_1\})$, so what we need to show is

$$\beta \cup \{P\} \vdash \textbf{assume } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P, P_1\}) \rightsquigarrow P_1 \Rightarrow P_2.$$

There are two cases: (a) $P = P_1$, and (b) $P \neq P_1$. In (a), the result follows from 1.25 and dilution, since $\Phi \cup \{P_1\} = \Phi \cup \{P, P_1\}$. If (b) holds, then, from the inductive hypothesis, 1.26 entails that

$$\beta \cup \{P\} \cup \{P_1\} \vdash RR(D_1, \Phi \cup \{P_1\} \cup \{P\}) \rightsquigarrow P_2$$

and hence

$$\beta \cup \{P\} \vdash \textbf{assume } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P, P_1\}) \rightsquigarrow P_1 \Rightarrow P_2$$

which is exactly what we wanted to show.

The reasoning for proofs by contradiction is similar. Specifically, when D is of the form

$$\textbf{suppose-absurd } P_1 \textbf{ in } D_1,$$

the assumption $\beta \vdash RR(D, \Phi) \rightsquigarrow Q$ translates to

$$\beta \vdash \textbf{suppose-absurd } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P_1\}) \rightsquigarrow \neg P_1 \tag{1.27}$$

which means that

$$\beta \cup \{P_1\} \vdash RR(D_1, \Phi \cup \{P_1\}) \rightsquigarrow \textbf{false}. \tag{1.28}$$

If $P = P_1$, then the desired conclusion

$$\beta \cup \{P\} \vdash RR(\textbf{suppose-absurd } P_1 \textbf{ in } D_1, \Phi \cup \{P\}) \rightsquigarrow Q = \neg P_1 \tag{1.29}$$

follows from 1.27 and dilution. If $P \neq P_1$, the the inductive hypothesis in tandem with 1.28 yield

$$\beta \cup \{P_1\} \cup \{P\} \vdash RR(D_1, \Phi \cup \{P_1\} \cup \{P\}) \rightsquigarrow \textbf{false} \tag{1.30}$$

hence $\beta \cup \{P\} \vdash$ **suppose-absurd** P_1 **in** $RR(D_1, \Phi \cup \{P, P_1\}) \rightsquigarrow \neg P_1$, which is the desired 1.29.

Finally, suppose that D is of the form $D_1; D_2$ and that $\beta \vdash RR(D, \Phi) \rightsquigarrow Q$, which is to say

$$\beta \vdash D_1'; D_2' \rightsquigarrow Q \tag{1.31}$$

where

$$D_1' = RR(D_1, \Phi) \tag{1.32}$$

$$D_2' = RR(D_2, \Phi \cup \{P_1\}) \tag{1.33}$$

$$\beta \vdash D_1' \rightsquigarrow P_1 \tag{1.34}$$

and

$$\beta \cup \{P_1\} \vdash D_2' \rightsquigarrow Q. \tag{1.35}$$

We need to prove $\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \rightsquigarrow Q$, i.e.,

$$\beta \cup \{P\} \vdash D_1''; D_2'' \rightsquigarrow Q \tag{1.36}$$

where

$$D_1'' = RR(D_1, \Phi \cup \{P\}) \tag{1.37}$$

and

$$D_2'' = RR(D_2, \Phi \cup \{P\} \cup \mathcal{C}(D_1'')). \tag{1.38}$$

On the basis of 1.32 and the inductive hypothesis, 1.34 implies that

$$\beta \cup \{P\} \vdash RR(D_1, \Phi \cup \{P\}) \rightsquigarrow P_1$$

i.e.,

$$\beta \cup \{P\} \vdash D_1'' \rightsquigarrow P_1 \tag{1.39}$$

so that

$$\mathcal{C}(D_1'') = P_1. \tag{1.40}$$

Likewise, by virtue of 1.33 and the inductive hypothesis, 1.35 implies that

$$\beta \cup \{P\} \cup \{P_1\} \vdash RR(D_2, \Phi \cup \{P_1\} \cup \{P\}) \rightsquigarrow Q. \tag{1.41}$$

From 1.38 and 1.40 we get $D_2'' = RR(D_2, \Phi \cup \{P\} \cup \{P_1\})$, so from 1.41,

$$\beta \cup \{P\} \cup \{P_1\} \vdash D_2'' \rightsquigarrow Q. \tag{1.42}$$

Finally, 1.36 follows from 1.39 and 1.42, hence $\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \rightsquigarrow Q$. This completes the induction. ■

Theorem 1.13 $D \rightarrowtail \mathfrak{P}(D)$.

Proof: We will prove that $D \rightarrowtail RR(D, \emptyset)$ by induction on D. When D is an atomic deduction, $RR(D, \emptyset) = D$, so the result is immediate since $\rightarrowtail$ is reflexive. When D is of the form

assume P **in** D_b,

$RR(D,\emptyset) =$ **assume** P **in** $RR(D_b,\{P\})$, so to show $D \rightarrowtail RR(D,\emptyset)$ we need to prove that if

$$\beta \vdash \textbf{assume } P \textbf{ in } D_b \rightsquigarrow P \Rightarrow Q \tag{1.43}$$

then

$$\beta \vdash \textbf{assume } P \textbf{ in } RR(D_b,\{P\}) \rightsquigarrow P \Rightarrow Q. \tag{1.44}$$

On the assumption that 1.43 holds, we have

$$\beta \cup \{P\} \vdash D_b \rightsquigarrow Q. \tag{1.45}$$

By the inductive hypothesis, $D_b \rightarrowtail RR(D_b,\emptyset)$, so from 1.45 we get

$$\beta \cup \{P\} \vdash RR(D_b,\emptyset) \rightsquigarrow Q$$

and by Lemma 1.12, $\beta \cup \{P\} \vdash RR(D_b,\{P\}) \rightsquigarrow Q$. Therefore,

$$\beta \vdash \textbf{assume } P \textbf{ in } RR(D_b,\{P\}) \rightsquigarrow P \Rightarrow Q$$

which is the desired 1.44.

Proofs by contradiction are handled similarly. In particular, suppose that D is of the form

$$\textbf{suppose-absurd } P \textbf{ in } D_b$$

and assume that $\beta \vdash$ **suppose-absurd** P **in** $D_b \rightsquigarrow \neg P$, for arbitrary β, so that

$$\beta \cup \{P\} \vdash D_b \rightsquigarrow \textbf{false}. \tag{1.46}$$

Inductively, $D_b \rightarrowtail RR(D_b,\emptyset)$, so 1.46 gives

$$\beta \cup \{P\} \vdash RR(D_b,\emptyset) \rightsquigarrow \textbf{false}.$$

Therefore, Lemma 1.12 yields

$$\beta \cup \{P\} \vdash RR(D_b,\{P\}) \rightsquigarrow \textbf{false}$$

and this implies $\beta \vdash$ **suppose-absurd** P **in** $RR(D_b,\{P\}) \rightsquigarrow \neg P$. We have thus shown that

$$\textbf{suppose-absurd } P \textbf{ in } D_b \rightarrowtail \textbf{suppose-absurd } P \textbf{ in } RR(D_b,\{P\})$$

which is to say $D \rightarrowtail RR(D,\emptyset)$.

Finally, suppose that D is of the form $D_1; D_2$ and that $\beta \vdash D_1; D_2 \rightsquigarrow Q$, so that

$$\beta \vdash D_1 \rightsquigarrow P \tag{1.47}$$

and

$$\beta \cup \{P\} \vdash D_2 \rightsquigarrow Q. \tag{1.48}$$

We have $RR(D,\emptyset) = D_1'; D_2'$, where

$$D_1' = RR(D_1,\emptyset) \tag{1.49}$$

and

$$D_2' = RR(D_2, \mathcal{C}(D_1')). \tag{1.50}$$

From the inductive hypothesis, $D_1 \rightarrowtail RR(D_1, \emptyset)$, hence from 1.47,

$$\beta \vdash RR(D_1, \emptyset) \leadsto P \tag{1.51}$$

so from 1.49,

$$\beta \vdash D_1' \leadsto P \tag{1.52}$$

and

$$\mathcal{C}(D_1') = P. \tag{1.53}$$

Likewise, $D_2 \rightarrowtail RR(D_2, \emptyset)$, so from 1.48,

$$\beta \cup \{P\} \vdash RR(D_2, \emptyset) \leadsto Q$$

and from Lemma 1.12,

$$\beta \cup \{P\} \vdash RR(D_2, \{P\}) \leadsto Q$$

which, from 1.50 and 1.53 means

$$\beta \cup \{P\} \vdash D_2' \leadsto Q. \tag{1.54}$$

Finally, from 1.52 and 1.54 we obtain $\beta \vdash D_1'; D_2' \leadsto Q$, and thus we infer $D \rightarrowtail RR(D, \emptyset) = D_1'; D_2'$. ■

Claim elimination

The third and final contracting transformation we will present is particularly simple: it eliminates all claims in non-tail positions. It is readily verified that all such claims are superfluous. For example, the claim in

$$D = \textbf{dn } \neg\neg A; \textbf{claim } B; \textbf{both } A, A$$

can be removed because $D \rightarrowtail \textbf{dn } \neg\neg A; \textbf{both } A, A$.

Claims in tail positions cannot in general be removed, since they serve as conclusions. One exception, however, occurs when the claim of some P is the last element of a thread whose immediately preceding element concludes P. In those cases the claim can be removed despite being in tail position. An example is

$$\textbf{dn } \neg\neg A; \textbf{both } A, B; \textbf{claim } A \wedge B.$$

Here the tail claim of $A \wedge B$ can be eliminated because it is derived by the immediately dominating deduction **both** A, B.

The following algorithm removes all claims in non-tail positions, as well as all extraneous tail claims of the sort discussed above:

```
$\mathfrak{C}(D) =$
  $match\ \ D$
      $\textbf{assume } P \textbf{ in } D_b \rightarrow \textbf{assume } P \textbf{ in } \mathfrak{C}(D_b)$
      $\textbf{suppose-absurd } P \textbf{ in } D_b \rightarrow \textbf{suppose-absurd } P \textbf{ in } \mathfrak{C}(D_b)$
      $D_1; D_2 \rightarrow$  $let\ \ D_1' = \mathfrak{C}(D_1)$
                          $D_2' = \mathfrak{C}(D_2)$
                    $in$
                      $claim?(D_1') \rightarrow D_2', claim?(D_2')\ \ and\ \ \mathcal{C}(D_1') = \mathcal{C}(D_2') \rightarrow D_1', D_1'; D_2'$
      $D \rightarrow D$
```

where *claim?*(D) returns true iff D is an application of **claim**. We have:

Lemma 1.14 **claim** $P; D \rightarrowtail D$. *Further,* $D;$**claim** $P \rightarrowtail D$ *whenever* $\mathcal{C}(D) = P$.

Using this lemma, a straightforward induction will show that $D \rightarrowtail \mathfrak{C}(D)$. Termination is immediate.

Theorem 1.15 $\mathfrak{C}$ *always terminates. In addition,* $D \rightarrowtail \mathfrak{C}(D)$.

Another property that will prove useful is the following:

Lemma 1.16 *Let* $D_1; \ldots; D_n; D_{n+1}$ *be a chain in* $\mathfrak{C}(D)$*,* $n > 0$*. Then* $\forall\, i \in \{1, \ldots, n\}$*,* D_i *is not a claim.*

Recall from 1.2 that the contracting phase of *simplify* is defined as

$$\mathit{contract} = \mathit{fp}\ \ (\mathfrak{C} \cdot \mathfrak{P} \cdot \mathfrak{U}).$$

For any given D, let us write $NT(D)$ to denote the number of non-trivial subdeductions of D, i.e., the number of subdeductions of D that are not claims. Define a quantity $Q(D)$ as the pair $(SZ(D), NT(D))$. A simple induction on D will show:

- $\mathfrak{U}\ \ D = D$ or $SZ(\mathfrak{U}\ \ D) < SZ(D)$;
- $\mathfrak{P}\ \ D = D$ or else $SZ(\mathfrak{P}\ \ D) < SZ(D)$ or $SZ(\mathfrak{P}\ \ D) = SZ(D)$ and $NT(\mathfrak{P}\ \ D) < NT(D)$;
- $\mathfrak{C}\ \ D = D$ or $SZ(\mathfrak{C}\ \ D) < SZ(D)$.

Therefore, writing $(a_1, b_1) <_{lex} (a_2, b_2)$ to mean that $a_1 < a_2$ or $a_1 = a_2$ and $b_1 < b_2$, we have:

Lemma 1.17 *For all D, either* $(\mathfrak{C} \cdot \mathfrak{P} \cdot \mathfrak{U})\ \ D = D$ *or else* $Q((\mathfrak{C} \cdot \mathfrak{P} \cdot \mathfrak{U})\ \ D) <_{lex} Q(D)$.

It follows that the fixed-point algorithm will eventually converge, since an infinitely long chain of distinct deductions $D_1, D_2, \ldots$ produced by repeated applications of $\mathfrak{C} \cdot \mathfrak{P} \cdot \mathfrak{U}$ would entail

$$Q(D_{i+1}) <_{lex} Q(D_i)$$

for all i, which is impossible since $<_{lex}$ is well-founded. It also follows from Lemma 1.17 that the size of the final result of *contract* will not be greater than the size of the original input. Finally, $D \rightarrowtail \mathit{contract}(D)$ follows from Theorem 1.11, Theorem 1.13, Theorem 1.15, and the transitivity of $\rightarrowtail$. We summarize:

Theorem 1.18 *The contraction procedure always terminates. In addition,* $D \rightarrowtail \mathit{contract}(D)$ *and* $SZ(\mathit{contract}(D)) \leq SZ(D)$.

1.4 Restructuring transformations

Scope maximization

The most fundamental restructuring transformation is *scope maximization*. Intuitively, this aims at making the conclusion of a subdeduction visible to as many subsequent parts of a deduction as possible. Scope can be limited in two ways: with bracketing (**begin-end** pairs), and with hypothetical deductions. We examine each case below.

Left-linear compositions

The first factor that can affect conclusion visibility is left-linear composition, namely, deductions of the form $(D_1; D_2); D_3$, where the conclusion of D_1 is only available to D_2. Such deductions are rare in practice because the natural threading style in $\mathcal{NDL}$ is right-associative (which is why composition associates to the right by default). When they occur, left-linear compositions can complicate our parsimony analysis. Consider, for instance, $D = (D_1; D_2); D_3$ where $\mathcal{C}(D_1) = \mathcal{C}(D_3)$. Algorithm $\mathfrak{P}$ might well find D to be repetition-free even though, intuitively, it is clear that D_3 unnecessarily duplicates the work of D_1. The problem is the limited scope of D_1: as long as D_2 does not replicate the conclusion of D_1 and D_3 the conclusion of $D_1; D_2$, i.e., the conclusion of D_2, then D will be deemed repetition-free. The problem can be avoided by right-associating D, thereby maximizing the scope of D_1. The algorithm $\mathfrak{RL}$ that we present below converts every subdeduction of the form $(D_1; D_2); D_3$ into $D_1; (D_2; D_3)$. Our proof that this is a safe transformation will be based on Lemma 1.19 below. The proof of the lemma is straightforward and omitted, but the intuition is important: in both cases D_1 is available to D_2, and D_2 to D_3, but in $D_1; (D_2; D_3)$ we *also* have D_1 available to D_3. So if $(D_1; D_2); D_3$ goes through, then certainly $D_1; (D_2; D_3)$ will do too.

Lemma 1.19 $(D_1; D_2); D_3 \rightarrowtail D_1; (D_2; D_3)$.

Let us say that a deduction D is *right-linear* iff $\mathit{Label}(D, u \oplus [1]) \neq ;$ for all $u \in \mathit{Dom}(D)$ such that $\mathit{Label}(D, u) = ;$. That is, D is right-linear iff it has no subdeductions of the form $(D_1; D_2); D_3$. The following is immediate:

Lemma 1.20 *If D is right-linear then so are* **assume** P **in** D *and* **suppose-absurd** P **in** D. *Moreover, if D_1 and D_2 are right-linear and D_1 is not a composite deduction then $D_1; D_2$ is right-linear.*

Algorithm $\mathfrak{RL}$ will transform any given D into a right-linear D' such that $D \rightarrowtail D'$:

$$\begin{array}{rcl}
\mathfrak{RL}(\textbf{assume } P \textbf{ in } D) & = & \textbf{assume } P \textbf{ in } \mathfrak{RL}(D) \\
\mathfrak{RL}(\textbf{suppose-absurd } P \textbf{ in } D) & = & \textbf{suppose-absurd } P \textbf{ in } \mathfrak{RL}(D) \\
 & & \textit{match } D_l \\
\mathfrak{RL}(D_l; D_r) & = & \quad D_1; D_2 \to \mathfrak{RL}(D_1; (D_2; D_r)) \\
 & & \quad \_ \to \mathfrak{RL}(D_l); \mathfrak{RL}(D_r) \\
\mathfrak{RL}(D) & = & D
\end{array}$$

For our termination proof, let us write $\mathit{SZ}(D)$ to denote the size of D, and let us define a quantity $\mathit{LSZ}(D)$ as follows: if D is of the form $D_1; D_2$ then $\mathit{LSZ}(D) = \mathit{SZ}(D_1)$; otherwise $\mathit{LSZ}(D) = 0$. It immediately follows:

Lemma 1.21 (a) *$\mathit{LSZ}((D_1; D_2); D_3) < \mathit{LSZ}(D_1; (D_2; D_3))$, and* (b) $\mathit{SZ}((D_1; D_2); D_3) = \mathit{SZ}(D_1; (D_2; D_3))$.

Theorem 1.22 $\mathfrak{RL}$ *always terminates.*

Proof: We claim that with each recursive call, the pair $(\mathit{SZ}(D), \mathit{LSZ}(D))$ strictly decreases lexicographically.[2] This can be seen by checking each recursive call: in the recursive calls of the first two lines, the size of D strictly decreases. In the recursive call $\mathfrak{RL}(D_1; (D_2; D_r))$ the size does not increase (Lemma 1.21, part (b)), while the quantity LSZ strictly decreases ((Lemma 1.21, part (a)). Finally, in both recursive calls in the next line, the size strictly decreases. ■

[2]Using the lexicographic extension of $<$ to pairs of natural numbers: (n_1, n_2) is smaller than (n'_1, n'_2) iff $n_1 < n'_1$ or else $n_1 = n'_1$ and $n_2 < n'_2$.

Theorem 1.23 $\mathfrak{RL}(D)$ *is right-linear. Furthermore,* $D \rightarrowtail \mathfrak{RL}(D)$.

Proof: Let us write $D_1 \prec D_2$ to mean that the pair $(SZ(D_1), LSZ(D_1))$ is lexicographically smaller than $(SZ(D_2), LSZ(D_2))$. We will use well-founded induction on the relation $\prec$, i.e., we will show that for all deductions D, if the result holds for every D' such that $D' \prec D$ then it also holds for D. We proceed by a case analysis of an arbitrary D. If D is an atomic deduction then $\mathfrak{RL}(D) = D$ and the result follows immediately. If D is a conditional deduction with hypothesis P and body D' then $\mathfrak{RL}(D) =$ **assume** P **in** $\mathfrak{RL}(D')$. Since $D' \prec D$, the inductive hypothesis entails that $\mathfrak{RL}(D')$ is right-linear and that $D' \rightarrowtail \mathfrak{RL}(D')$. The same reasoning is used for proofs by contradiction. The result now follows from Lemma 1.20 and Lemma 1.8.

Finally, suppose that D is of the form $D_l; D_r$. Then either D_l is of the form $D_1; D_2$, or not. In the first case we have

$$\mathfrak{RL}(D) = \mathfrak{RL}(D_1; (D_2; D_r)) \tag{1.55}$$

and since $D = (D_1; D_2); D_r \succ D_1; (D_2; D_r)$, we conclude inductively that

(i) $\mathfrak{RL}(D_1; (D_2; D_r))$ is right linear, and

(ii) $D_1; (D_2; D_r) \rightarrowtail \mathfrak{RL}(D_1; (D_2; D_r))$.

Thus the conclusion that $\mathfrak{RL}(D)$ is right-linear follows from (i) and 1.55, while

$$D \rightarrowtail \mathfrak{RL}(D) = \mathfrak{RL}(D_1; (D_2; D_r))$$

follows from (ii) and the transitivity of $\rightarrowtail$, since $D \rightarrowtail D_1; (D_2; D_r)$ from Lemma 1.19. If D is not of the form $D_1; D_2$ then

$$\mathfrak{RL}(D) = \mathfrak{RL}(D_l); \mathfrak{RL}(D_r) \tag{1.56}$$

and since $D_l \prec D$, $D_r \prec D$, the inductive hypothesis entails that (i) $\mathfrak{RL}(D_l)$ and $\mathfrak{RL}(D_r)$ are right-linear, and (ii) $D_l \rightarrowtail \mathfrak{RL}(D_l)$, $D_r \rightarrowtail \mathfrak{RL}(D_r)$. Because D_l is not a composite deduction, neither is $\mathfrak{RL}(D_l)$ (a necessary condition for $\mathfrak{RL}(D)$ to be composite is that D be composite), hence it follows from part (b) of Lemma 1.20 and 1.56 that $\mathfrak{RL}(D)$ is right-linear. Further, $D \rightarrowtail \mathfrak{RL}(D)$ follows from (ii) and Lemma 1.8. This concludes the case analysis and the inductive argument. ■

Hypothetical deductions

The second case of undue scope limitation arises in hypothetical deductions. Consider a hypothetical deduction with body D_b and hypothesis P. If D is a subdeduction of D_b then its scope cannot extend beyond D_b. But this need not be the case if D is not strictly dependent on the hypothesis P. If there is no such dependence, then D is unnecessarily restricted by being inside D_b. Its scope should be maximized by *hoisting* it outside D_b. As a simple example, consider

```
assume $B$ in
  begin
    double-negation $\neg\neg A$;
    both $A, B$
  end
```

Here the subdeduction **double-negation** $\neg\neg A$ makes no use of the hypothesis B, and therefore it is appropriate to pull it outside, resulting in

```
double-negation $\neg\neg A$;
assume $B$ in
  both $A, B$
```

This deduction is observationally equivalent to the first one, and has a cleaner structure that better reflects the various logical dependencies. Besides increased clarity, hoisting will greatly facilitate our repetition analysis later on. Repetitions are much easier to detect and eliminate when they are in the same scope. Consider, for instance, the deduction

```
assume $B$ in
  begin
    double-negation $\neg\neg A$;
    both $A, B$
  end;
left-and $A \wedge C$;
both $A, B \Rightarrow A \wedge B$
```

In view of **double-negation** $\neg\neg A$, the deduction **left-and** $A \wedge C$ is superfluous, but this is not easy to determine mechanically because the former deduction lies inside the scope of the hypothesis B. More importantly, neither deduction can be safely eliminated as things stand, even though it is clearly extraneous to have both of them. If we eliminated the **double-negation** then the **assume** might fail; while if we eliminated the **left-and**, the composition might fail. But if we hoist the double negation outside of the **assume**, resulting in

```
double-negation $\neg\neg A$;
assume $B$ in
  both $A, B$;
left-and $A \wedge C$;
both $A, B \Rightarrow A \wedge B$
```

then the repetition becomes much easier to detect, and the **left-and** can be confidently eliminated.

In what follows we will be dealing with lists of deductions $[D_1, \ldots, D_n]$. We will use the letter Δ to denote such lists. For a non-empty list $\Delta = [D_1, \ldots, D_n]$, $n > 0$, we define $\overline{\Delta}$ as the thread $D_1; \ldots; D_n$. The following will come handy later:

Lemma 1.24 $\overline{\Delta_1 \oplus \Delta_2} = \overline{\Delta_1}; \overline{\Delta_2}$

We adopt the convention that when Δ is empty the expresssion $\overline{\Delta}; D$ stands for D.

The algorithm H in Figure 1.6 examines a right-linear thread $D = D_1; \ldots; D_n$ (we make the simplifying convention that we might have $n = 1$, in which case D_1 will not be composite, since we are assuming that D is right-linear) and pulls out every D_i that is not transitively dependent on a set of assumptions Φ. Each hoisted D_i is replaced in-place in D by the trivial deduction **claim** $\mathcal{C}(D_i)$. Specifically, $H(D, \Phi)$ returns a triple (D', Ψ, Δ), where

- D' is obtained from D by replacing every D_i that does not transitively depend on Φ by $\mathcal{C}(D_i)$.
- $\Psi \supseteq \Phi$ is monotonically obtained from Φ by incorporating the conclusions of those deductions D_j that do depend (transitively) on Φ. This is essential in order to handle transitive dependence.

$$\begin{array}{rcl}
 & & \mathit{let}\ (D'_1, \Phi_1, \Delta_1) = H(D_1, \Phi) \\
 & & \quad (D'_2, \Phi_2, \Delta_2) = H(D_2, \Phi_1) \\
H(D_1; D_2, \Phi) & = & \mathit{in} \\
 & & \quad (D'_1; D'_2, \Phi_2, \Delta_1 \oplus \Delta_2) \\
 & & \\
H(D, \Phi) & = & \mathit{FA}(D) \cap \Phi = \emptyset\,? \to (\mathbf{claim}\ \mathcal{C}(D), \Phi, [D]),\ (D, \Phi \cup \{\mathcal{C}(D)\}, [])
\end{array}$$

Figure 1.6: The kernel of the hoisting algorithm.

- Δ is a list $[D_{i_1}, \ldots, D_{i_k}]$, $1 \le i_j \le n$, $j = 1, \ldots, k \ge 0$, of those deductions that do not depend on Φ. The order is important for preserving dominance constraints: we have $i_a < i_b$ for $a < b$, since, e.g., D_5 and D_8 might not be dependent on Φ, but D_8 might depend on D_5. Accordingly, Δ should respect the original ordering.

As Theorem 1.30 will prove, the idea is that we will have $D \rightarrowtail \overline{\Delta}; D'$. The thread $D_1; \cdots; D_n$ should be thought of as the body of a hypothetical deduction with hypothesis P, and Φ should be thought of as $\{P\}$. Then if $H(D_1; \ldots; D_n, \Phi) = (D', \Psi, \Delta)$, D' will be the new body of the hypothetical deduction, and the thread $\overline{\Delta}$ will comprise the hoisted deductions, with a dominance relation that respects the original ordering $1, \ldots, n$.

Lemma 1.25 *Let $H(D_1, \Phi_1) = (D_2, \Phi_2, \Delta)$. Then for all $D \in \Delta$,* (a) *$\Phi_1 \cap \mathit{FA}(D) = \emptyset$, and* (b) *$D$ is not a composition.*

Proof: By induction on D_1. Suppose first that D_1 is not composite. There are two cases: either $\mathit{FA}(D_1) \cap \Phi_1 = \emptyset$ or not. If not, then $\Delta = []$ so the result holds vacuously. If $\mathit{FA}(D_1) \cap \Phi_1 = \emptyset$ then $\Delta = [D_1]$, so the result holds by supposition. Finally, if D_1 is of the form $D_l; D_r$ then $\Delta = \Delta_l \oplus \Delta_r$, where $H(D_l, \Phi_1) = (D'_l, \Phi_l, \Delta_l)$ and $H(D_r, \Phi_l) = (D'_r, \Phi_r, \Delta_r)$. Inductively,

$$\forall D \in \Delta_l, \Phi_1 \cap \mathit{FA}(D) = \emptyset \tag{1.57}$$

and

$$\forall D \in \Delta_r, \Phi_l \cap \mathit{FA}(D) = \emptyset \tag{1.58}$$

while every D in Δ_l and Δ_r is a non-composition. Since $\Phi_1 \subseteq \Phi_l$, 1.58 entails

$$\forall D \in \Delta_r, \Phi_1 \cap \mathit{FA}(D) = \emptyset \tag{1.59}$$

Part (a) now follows from 1.57 and 1.59 since $\Delta = \Delta_l \oplus \Delta_r$, while (b) follows directly from the inductive hypotheses. ■

We will also need the following four results, whose proofs are simple and omitted:

Lemma 1.26 *Let $H(D_1, \Phi_1) = (D_2, \Phi_2, \Delta)$. If D_1 is right-linear then D_2 is right-linear, and every $D \in \Delta$ is right-linear too.*

Lemma 1.27 *Let $(D', \Psi, \Delta) = H(D, \Phi)$. Then either*

1. *$D' = D$; or else*

2. *D' is a claim; or*
3. *D is a chain $D_1, \ldots, D_n, D_{n+1}$ and D' is a chain $D'_1, \ldots, D'_n, D'_{n+1}$, where for all i, either $D'_i = D_i$ or else D'_i is a claim.*

Lemma 1.28 *If $\mathcal{C}(D) \notin FA(D_i)$ for $i = 1, \ldots, n$ then*

$$D; D_1; \ldots; D_n; D' \rightarrowtail D_1; \ldots; D_n; D; D'.$$

Lemma 1.29 *$P; D_1; \ldots; D_n; D \rightarrowtail D_1; \ldots; D_n; P; D$.*

Theorem 1.30 *If D is right-linear and $(D', \Psi, \Delta) = H(D, \Phi)$ then $D \rightarrowtail \overline{\Delta}; D'$.*
Proof: By induction on D. Suppose first that D is not a composition. Then either $FA(D) \cap \Phi = \emptyset$ or not. If not, then $D' = D$ and $\Delta = []$, so the result is immediate. If $FA(D) \cap \Phi = \emptyset$ then $D' = \mathcal{C}(D)$ and $\Delta = [D]$, so again the result follows directly. Suppose next that D is of the form $D_1; D_2$. Then, letting

$$(D'_1, \Phi_1, \Delta_1) = H(D_1, \Phi) \tag{1.60}$$

and

$$(D'_2, \Phi_2, \Delta_2) = H(D_2, \Phi_1) \tag{1.61}$$

we have $D' = D'_1; D'_2$ and $\Delta = \Delta_1 \oplus \Delta_2$, so we have to show

$$D \rightarrowtail \overline{\Delta_1 \oplus \Delta_2}; D'_1; D'_2. \tag{1.62}$$

From 1.60, 1.61, and the inductive hypothesis, we have

$$D_1 \rightarrowtail \overline{\Delta_1}; D'_1 \tag{1.63}$$

and

$$D_2 \rightarrowtail \overline{\Delta_2}; D'_2 \tag{1.64}$$

Therefore,

$$D = D_1; D_2 \rightarrowtail \overline{\Delta_1}; D'_1; \overline{\Delta_2}; D'_2 \tag{1.65}$$

Now since we are assuming that D is right-linear, D_1 cannot be composite, hence again we distinguish two cases: $FA(D_1) \cap \Phi = \emptyset$, or not. If the latter holds then $D'_1 = D_1, \Phi_1 = \Phi \cup \{\mathcal{C}(D_1)\}$, and $\Delta_1 = []$. Now by 1.61 and Lemma 1.25 we have that, for every $D_x \in \Delta_2$, $\Phi_1 \cap FA(D_x) = \emptyset$, and since $\mathcal{C}(D_1) \in \Phi_1$, this means that $\mathcal{C}(D_1) \notin FA(D_x)$. Hence, from Lemma 1.28,

$$D'_1; \overline{\Delta_2}; D'_2 \rightarrowtail \overline{\Delta_2}; D'_1; D'_2$$

and thus

$$\overline{\Delta_1}; D'_1; \overline{\Delta_2}; D'_2 \rightarrowtail \overline{\Delta_1}; \overline{\Delta_2}; D'_1; D'_2. \tag{1.66}$$

On the other hand, if $FA(D_1) \cap \Phi = \emptyset$ then $D'_1 = \mathcal{C}(D_1)$, so by Lemma 1.29 we have

$$D'_1; \overline{\Delta_2}; D'_2 \rightarrowtail \overline{\Delta_2}; D'_1; D'_2$$

and hence 1.66 follows again. Thus we have shown that in either case 1.66 holds, and since $\overline{\Delta_1 \oplus \Delta_2} = \overline{\Delta_1} \oplus \overline{\Delta_2}$, it now follows from 1.65, 1.66, and the transitivity of $\rightarrowtail$ that

$$D \rightarrowtail \overline{\Delta_1 \oplus \Delta_2}; D'_1; D'_2$$

which is 1.62, exactly what we wanted to show. This completes the induction. ■

Theorem 1.31 *If D is right-linear and $H(D, \{P\}) = (D', \Phi, \Delta)$ then*

(a) $\textbf{assume } P \textbf{ in } D \rightarrowtail \overline{\Delta}; \textbf{assume } P \textbf{ in } D'$*; and*

(b) $\textbf{suppose-absurd } P \textbf{ in } D \rightarrowtail \overline{\Delta}; \textbf{suppose-absurd } P \textbf{ in } D'$.

Proof: We prove (a); the proof of (b) is entirely analogous. First we note that, by Theorem 1.30, $D \rightarrowtail \overline{\Delta}; D'$, therefore, by Lemma 1.8,

$$\textbf{assume } P \textbf{ in } D \rightarrowtail \textbf{assume } P \textbf{ in } \overline{\Delta}; D'. \tag{1.67}$$

We now proceed by induction on the structure of Δ. When Δ is the empty list, 1.67 becomes

$$\textbf{assume } P \textbf{ in } D \rightarrowtail \textbf{assume } P \textbf{ in } D' = \overline{\Delta}; \textbf{assume } P \textbf{ in } D'.$$

When Δ is of the form $D_1{::}\Delta_1$, 1.67 becomes

$$\textbf{assume } P \textbf{ in } D \rightarrowtail \textbf{assume } P \textbf{ in } \overline{D_1{::}\Delta_1}; D' = \textbf{assume } P \textbf{ in } D_1; \overline{\Delta_1}; D'. \tag{1.68}$$

By Lemma 1.25, $\{P\} \cap FA(D_1) = \emptyset$, so Theorem 1.10 yields

$$\textbf{assume } P \textbf{ in } D_1; \overline{\Delta_1}; D' \rightarrowtail D_1; \textbf{assume } P \textbf{ in } \overline{\Delta_1}; D'. \tag{1.69}$$

By Lemma 1.25 and Lemma 1.26, every deduction in Δ_1 is right-linear and not a composition, and D' is right-linear as well, hence, using the second part of Lemma 1.20, a straightforward induction on the length of Δ_1 will show that $\overline{\Delta_1}; D'$ is right-linear. This means that the inductive hypothesis applies, and yields

$$\textbf{assume } P \textbf{ in } \overline{\Delta_1}; D' \rightarrowtail \overline{\Delta_1}; \textbf{assume } P \textbf{ in } D'$$

so, since $D_1 \rightarrowtail D_1$, Lemma 1.8 gives

$$D_1; \textbf{assume } P \textbf{ in } \overline{\Delta_1}; D' \rightarrowtail D_1; \overline{\Delta_1}; \textbf{assume } P \textbf{ in } D' = \overline{\Delta}; \textbf{assume } P \textbf{ in } D'. \tag{1.70}$$

Finally, from 1.68, 1.69, 1.70, and the transitivity of $\rightarrowtail$ we conclude

$$\textbf{assume } P \textbf{ in } D \rightarrowtail \overline{\Delta}; \textbf{assume } P \textbf{ in } \Delta'$$

and the induction is complete. ■

As an illustration of the algorithm, let D be the deduction

1. **modus-ponens** $A \Rightarrow B \wedge C, A$;
2. **double-negation** $\neg\neg E$;
3. **left-and** $B \wedge C$;
4. **right-either** F, E;
5. **both** $B, F \vee E$

and consider the call $H(D, \{A\})$. Let D_1–D_5 refer to the deductions in lines 1–5, respectively. Since D is composite, the first clause of the algorithm will be chosen, so the first recursive call will be $H(D_1, \{A\})$, which, since D_1 is not composite and $FA(D_1) \cap \{A\} \neq \emptyset$, will yield the result

$(D_1, \{A, B \wedge C\}, [])$. The second recursive call is $H(D_2; D_3; D_4; D_5, \{A, B \wedge C\})$. This in turn gives rise to the recursive calls $H(D_2, \{A, B \wedge C\})$, which returns

$$(\textbf{claim } E, \{A, B \wedge C\}, [\textbf{double-negation } \neg\neg E]),$$

and $H(D_3; D_4; D_5, \{A, B \wedge C\})$. The latter will spawn $H(D_3, \{A, B \wedge C\})$, which will produce

$$(D_3, \{A, B \wedge C, B\}, []),$$

and $H(D_4; D_5, \{A, B \wedge C, B\})$. In the same fashion, the latter will spawn $H(D_4, \{A, B \wedge C, B\})$, which will return

$$(\textbf{claim } F \vee E, \{A, B \wedge C, B\}, [\textbf{right-either } F, E]),$$

and $H(D_5, \{A, B \wedge C, B\})$, which will produce $(D_5, \{A, B \wedge C, B, B \wedge (F \vee E)\}, [])$. Moving up the recursion tree will eventually yield the final result (D', Ψ, Δ), where D' is the deduction

1.**modus-ponens** $A \Rightarrow B \wedge C, A$;
2.**claim** E;
3.**left-and** $B \wedge C$;
4.**claim** $F \vee E$;
5.**both** $B, F \vee E$

while Ψ is the set $\{A, B \wedge C, B, B \wedge (F \vee E)\}$ and Δ is the list

$$[\textbf{double-negation } \neg\neg E, \textbf{right-either } F, E].$$

Thus $\overline{\Delta}; D'$ is the deduction

double-negation $\neg\neg E$;
right-either F, E;

---

modus-ponens $A \Rightarrow B \wedge C, A$;
claim E;
left-and $B \wedge C$;
claim $F \vee E$;
both $B, F \vee E$

The horizontal line demarcates the hoisted deductions from D'.

If D were the body of a hypothetical deduction with hypothesis A, then the result of the hoisting would be

$$\overline{\Delta}; \textbf{assume } A \textbf{ in } D'$$

namely,

```
double-negation ¬¬E;
right-either F, E;
assume A in
  begin
    modus-ponens A ⇒ B ∧ C, A;
    claim E;
    left-and B ∧ C;
    claim F ∨ E;
    both B, F ∨ E
  end
```

A subsequent contracting transformation to remove claims (algorithm $\mathfrak{C}$) would result in

```
**double-negation** $\neg\neg E$;
**right-either** $F, E$;
**assume** $A$ **in**
  **begin**
    **modus-ponens** $A \Rightarrow B \wedge C, A$;
    **left-and** $B \wedge C$;
    **both** $B, F \vee E$
  **end**
```

The hoisting algorithm should be applied to every hypothetical deduction contained in a given D. This must be done in stages and in a bottom-up direction in order for hoisted inferences to "bubble" as far up as possible (to maximize their scope). Specifically, let D be a given deduction. The hoisting will proceed in stages $i = 1, \ldots, n, \ldots$, where we begin with $D_1 = D$. At each stage i we replace certain *candidate* hypothetical subdeductions of D_i by new deductions, and the result we obtain from these replacements becomes D_{i+1}. We keep going until we reach a fixed point, i.e., until $D_{i+1} = D_i$.

At each point in the process every hypothetical subdeduction of D_i is either *marked*, indicating that its body has already been processed, or unmarked. An invariant we will maintain throughout is that a marked hypothetical subdeduction will never contain unmarked hypothetical deductions; this will be enforced by the way in which we will be choosing our candidates, and will ensure that hoisting proceeds in a bottom-up direction. Initially, all hypothetical subdeductions of $D_1 = D$ are unmarked. On stage i, an unmarked hypothetical subdeduction of D_i is a candidate for hoisting iff it is as deep as possible, i.e., iff it does not itself contain any unmarked hypothetical subdeductions. For each such candidate $D_c =$ **assume** P **in** D_b (or $D_c =$ **suppose-absurd** P **in** D_b) occurring in position $u \in Dom(D_i)$, we compute $(D'_b, \Psi, \Delta) = H(D_b, \{P\})$, and we replace D_c in position u of D_i by $\overline{\Delta}$; $\underline{\textbf{assume}}$ P **in** D'_b (or $\overline{\Delta}$; $\underline{\textbf{suppose-absurd}}$ P **in** D'_b), where the **assume** (or **suppose-absurd**) is now marked to indicate that its body D'_b has been combed bottom-up and we are thus finished with it—it can no longer serve as a candidate. The deduction we obtain from D_i by carrying out these replacements becomes D_{i+1}. One pitfall to be avoided: the replacements might introduce left-linear subdeductions in D_{i+1}. Algorithm H, however, expects its argument to be right-linear, so after the replacements are performed we need to apply $\mathfrak{RL}$ to D_{i+1} before continuing on to the next stage.

Algorithm *Hoist* below replaces every candidate hypothetical subdeduction of a given D in the manner discussed above and marks the processed subdeduction:

```
Hoist(D) = match D
    **assume** $P$ **in** $D_b \rightarrow$
      *Is every* **assume** *and* **suppose-absurd** *within* $D_b$ *marked?* $\rightarrow$
        *let* $(D'_b, \_, \Delta) = H(D_b, \{P\})$
        *in*
          $\overline{\Delta}$; $\underline{\textbf{assume}}$ $P$ **in** $D'_b$,
        **assume** $P$ **in** $Hoist(D_b)$
    **suppose-absurd** $P$ **in** $D_b \rightarrow$
      *Is every* **assume** *and* **suppose-absurd** *within* $D_b$ *marked?* $\rightarrow$
        *let* $(D'_b, \_, \Delta) = H(D_b, \{P\})$
        *in*
          $\overline{\Delta}$; $\underline{\textbf{suppose-absurd}}$ $P$ **in** $D'_b$,
```

$$
\begin{array}{l}
\quad\quad \textbf{suppose-absurd } P \textbf{ in } \mathit{Hoist}(D_b) \\
D_1; D_2 \rightarrow \mathit{Hoist}(D_1); \mathit{Hoist}(D_2) \\
D \rightarrow D
\end{array}
$$

Using Theorem 1.31 and Lemma 1.8, a straightforward induction on D will prove the following result:

Theorem 1.32 *If D is right-linear then $D \rightarrowtail \mathit{Hoist}(D)$.*

We can now formulate our final scope-maximization transformation as follows:

$$\mathfrak{MS}\ D = \mathit{fp}\ (\mathfrak{RL} \cdot \mathit{Hoist})\ (\mathfrak{RL}\ D)$$

where *fp* is as defined in Section 1.1. That $\mathfrak{MS}$ always terminates follows from the fact that *Hoist* does not introduce any additional hypothetical deductions, and either outputs the same result unchanged (a fixed point) or a deduction with at least one more hypothetical deduction marked. Since any deduction only has a finite number of hypothetical subdeductions, this means that $\mathfrak{MS}$ will eventually converge to a fixed point. Further, $D \rightarrowtail \mathfrak{MS}(D)$ follows from the corresponding property of $\mathfrak{RL}$, from Theorem 1.32, and from the transitivity of the $\rightarrowtail$ relation. The right-linearity of the result follows directly from the definition of $\mathfrak{MS}$. We summarize:

Theorem 1.33 (a) $\mathfrak{MS}$ *always terminates;* (b) $\mathfrak{MS}(D)$ *is right-linear;* (c) $D \rightarrowtail \mathfrak{MS}(D)$.

We close by addressing the question of whether this restructuring algorithm might ever increase the size of a deduction. Since $\mathfrak{MS}$ works by repeatedly applying the composition of *Hoist* with $\mathfrak{RL}$, it will follow that $\mathfrak{MS}$ preserves the size of its argument if both $\mathfrak{RL}$ and *Hoist* do. This is readily verified for $\mathfrak{RL}$; we have $\mathit{SZ}(\mathfrak{RL}(D)) = \mathit{SZ}(D)$ for all D. Consider now the hoisting transformation H, which is the core of *Hoist*. When *Hoist* applies H to the body of a hypothetical deduction, say **assume** P **in** D_b, thereby obtaining a new deduction $\overline{\Delta}$; **assume** P **in** D'_b, the new part $\overline{\Delta}$ is obtained by trimming down the body D_b, so, intuitively, we should have $\mathit{SZ}(D_b) = \mathit{SZ}(D'_b) + \mathit{SZ}(\overline{\Delta})$. But that is not quite true because the new body D'_b might contain some claims where the hoisted deductions used to be, and those claims will cause the size of the result to be somewhat larger than that of the original. However, most such claims will be subsequently removed by the claim-elimination algorithm presented earlier, and this will rebalance the final size—even in the worst-case scenario in which the hoisting did not expose any new contraction opportunities. This is evinced by Lemma 1.27: claims inserted in D'_b in non-tail positions will be eliminated by $\mathfrak{C}$, as guaranteed by Lemma 1.16.

There is only one exception, again as prescribed by Lemma 1.27: when the new body D'_b is a chain of the form $D_1; \ldots; D_n$, $n \geq 1$, and the last element of the thread, D_n, is a newly inserted claim. Such a claim, being in a tail position, will *not* be removed by the claim-elimination algorithm. As a simple example, consider

$$D = \textbf{assume } A \textbf{ in double-negation } \neg\neg B. \tag{1.71}$$

Here the body does not depend on the hypothesis A, so hoisting it outside results in the deduction

$$\textbf{double-negation } \neg\neg B; \textbf{assume } A \textbf{ in claim } B$$

which is slightly larger than the original 1.71. But this minor wrinkle is easily rectified using **cond** (or **neg**, in the case of **suppose-absurd**). Specifically, by the way H is defined, a trivial deduction **claim** Q will be inserted in the last slot of D'_b (viewing D'_b as a chain of one or more elements) iff the last element of the produced list Δ is a deduction whose conclusion is Q. Therefore, in that case, instead of producing $\overline{\Delta}$; **assume** P **in** D'_b we may simply output $\overline{\Delta}$; **cond** $P, \mathcal{C}(\overline{\Delta})$; or, in the case of proofs by contradiction, $\overline{\Delta}$; **neg** P. Accordingly, we modify *Hoist* by replacing the line $\overline{\Delta}; \underline{\textbf{assume}}\ P$ **in** D'_b, by

$$\Delta \neq [] \ \textit{ and } \ \mathcal{C}(\overline{\Delta}) = \mathcal{C}(D'_b)\,? \to \overline{\Delta}; \textbf{cond}\ P, \mathcal{C}(D'_b),\ \overline{\Delta}; \underline{\textbf{assume}}\ P\ \textbf{in}\ D'_b,$$

and the line $\overline{\Delta}; \underline{\textbf{suppose-absurd}}\ P$ **in** D'_b, by

$$\Delta \neq [] \ \textit{ and } \ \mathcal{C}(\overline{\Delta}) = \mathcal{C}(D'_b)\,? \to \overline{\Delta}; \textbf{neg}\ P,\ \overline{\Delta}; \underline{\textbf{suppose-absurd}}\ P\ \textbf{in}\ D'_b.$$

It is readily verified that this change does not affect Theorem 1.33, yet it ensures that all claims inserted by H will be subsequently removed during the contraction phase.

Global transformations of hypothetical deductions

The hoisting algorithm is a focused, local transformation: we delve inside a given deduction D and work on subdeductions of the form **assume** P **in** D_b or **suppose-absurd** P **in** D_b, taking into account only the hypothesis P and the body D_b. We do not utilize any knowledge from a wider context. More intelligent transformations become possible if we look at the big picture, namely, at how P and D_b relate to other parts of the enclosing deduction D. In this section we will present three such transformations, $\mathfrak{A}_1$, $\mathfrak{A}_2$, and $\mathfrak{A}_3$. All three of them perform a global analysis of a given deduction D and replace every hypothetical subdeduction D' of it by some other deduction D'' (where we might have $D'' = D'$). These transformations expect their input deductions to have been processed by $\mathfrak{MS}$, but their output deductions might contain left-linear compositions or hoisting possibilities that were not previously visible. It is for this reason that their composition must be interleaved with the scope-maximization procedure $\mathfrak{MS}$, as specified in 1.3 (or 1.4).

The first transformation, $\mathfrak{A}_1$, targets every hypothetical subdeduction of D of the form $D' =$ **assume** P **in** D_b whose hypothesis P is a free assumption of D, i.e., such that $P \in FA(D)$. Clearly, D can only be successfully evaluated in an assumption base that contains P (Theorem 1.4). But if we must evaluate D in an assumption base that contains P, then there is no need to hide D_b behind that hypothesis; we can pull it outside. Accordingly, this analysis will replace D' by the composition $D'' = D_b$; **cond** $P, \mathcal{C}(D_b)$. Thus the final conclusion is unaffected (it is still the conditional $P \Rightarrow \mathcal{C}(D_b)$), but the scope of D_b is enlarged. An analogous transformation is performed for proofs by contradiction. Specifically, we define:

```
$\mathfrak{A}_1(D) = T(D)$
where
$T(\textbf{assume}\ P\ \textbf{in}\ D_b) =$
    $\textit{let}\ D'_b = T(D_b)$
    $\textit{in}$
      $P \in FA(D)\,? \to D'_b; \textbf{cond}\ P, \mathcal{C}(D'_b),\ \textbf{assume}\ P\ \textbf{in}\ D'_b$
$T(\textbf{suppose-absurd}\ P\ \textbf{in}\ D_b) =$
    $\textit{let}\ D'_b = T(D_b)$
    $\textit{in}$
      $P \in FA(D)\,? \to D'_b; \textbf{neg}\ P,\ \textbf{suppose-absurd}\ P\ \textbf{in}\ D'_b$
$T(D_l; D_r) = T(D_l); T(D_r)$
$T(D) = D$
```

Note that we first process D_b recursively and then pull it out, since D_b might itself contain hypothetical deductions with a free assumption as a hypothesis. For example, if D is the deduction

```
assume $A$ in
 begin
  both $A, A$;
  assume $B$ in
    both $B, A \wedge A$
 end;
both $A, B$;
both $A \wedge B, A \Rightarrow B \Rightarrow B \wedge A \wedge A$
```

where both conditional deductions have free assumptions as hypotheses (A and B) then $\mathfrak{A}_1(D)$ will be:

```
begin
  begin
    both $A, A$;
    both $B, A \wedge A$;
    cond $B, B \wedge A \wedge A$
  end;
  cond $A, B \Rightarrow B \wedge A \wedge A$
end;
both $A, B$;
both $A \wedge B, A \Rightarrow B \Rightarrow B \wedge A \wedge A$
```

Observe that the output deduction is heavily skewed to the left (when viewed as a tree). After a pass of the right-linearization algorithm, we will eventually obtain the following:

```
both $A, A$;
both $B, A \wedge A$;
cond $B, B \wedge A \wedge A$;
cond $A, B \Rightarrow B \wedge A \wedge A$;
both $A, B$;
both $A \wedge B, A \Rightarrow B \Rightarrow B \wedge A \wedge A$
```

A straightforward induction will show:

Lemma 1.34 $\mathfrak{A}_1$ *terminates. Moreover,* $D \rightarrowtail \mathfrak{A}_1(D)$ *and* $SZ(\mathfrak{A}_1(D)) \leq SZ(D)$.

The two remaining transformations turn not on whether the hypothesis of a conditional deduction is a free assumption, but on whether it is deduced at some prior or subsequent point. For the second transformation, $\mathfrak{A}_2$, suppose that during our evaluation of D we come to a conditional subdeduction $D' =$ **assume** P **in** D_b whose hypothesis P either has already been established or else has already been hypothetically postulated (i.e., D' is itself nested within an **assume** with hypothesis P). Then we may again pull D_b out, replacing D' by the composition $D'' = D_b;$ **assume** P **in** $\mathcal{C}(D_b)$. (More precisely, just as in $\mathfrak{A}_1$, we first have to process D_b recursively before hoisting it.) A similar transformation is possible for proofs by contradiction.

To motivate this transformation, consider the following deduction:

```
left-and $\neg\neg A \wedge C$;
assume $\neg\neg A$ in
 begin
```

```
    dn $\neg\neg A$;
    both $A, B$
  end;
modus-ponens $\neg\neg A \Rightarrow A \wedge B, \neg\neg A$
```

This deduction illustrates one of the detours we discussed earlier, whereby Q is derived by first inferring P, then $P \Rightarrow Q$, and then using **modus-ponens** on $P \Rightarrow Q$ and P. The detour arises because the hypothesis P is in fact deducible, and hence there is no need for the implication $P \Rightarrow Q$ and the **modus-ponens**. We can simply deduce P and then directly perform the reasoning of the body of the hypothetical deduction. Thus we arrive at the following algorithm:

```
$\mathfrak{A}_2(D) = T(D, \emptyset)$
where
$T(D, \Phi) =$ match $D$
              assume $P$ in $D_b \rightarrow$
                 $P \in \Phi \rightarrow$ let $D'_b = T(D_b, \Phi)$
                                in
                                  $D'_b$; cond $P, \mathcal{C}(D'_b)$,
                                assume $P$ in $T(D_b, \Phi \cup \{P\})$
              suppose-absurd $P$ in $D_b \rightarrow$
                 $P \in \Phi \rightarrow$ let $D'_b = T(D_b, \Phi)$
                                in
                                  $D'_b$; neg $P$,
                                suppose-absurd $P$ in $T(D_b, \Phi \cup \{P\})$
              $D_1; D_2 \rightarrow$ let $D'_1 = T(D_1, \Phi)$
                            in
                              $D'_1; T(D_2, \Phi \cup \{\mathcal{C}(D_1)\})$
              $D \rightarrow D$
```

Applying this algorithm to the foregoing example would yield:

```
left-and $\neg\neg A \wedge C$;
begin
  begin
    dn $\neg\neg A$;
    both $A, B$
  end;
  cond $\neg\neg A, A \wedge B$
end;
modus-ponens $\neg\neg A \Rightarrow A \wedge B, \neg\neg A$
```

Passing this on to the scope-maximization procedure and then to the contraction algorithm will produce the final result:

```
left-and $\neg\neg A \wedge C$;
dn $\neg\neg A$;
both $A, B$
```

We can establish the soundness of this algorithm in two steps. First, we can prove by induction on D that if $\beta \vdash T(D, \Phi) \rightsquigarrow Q$ then $\beta \cup \{P\} \vdash T(D, \Phi \cup \{P\}) \rightsquigarrow Q$. Then, using this lemma, an induction on D will show that $D \rightarrowtail T(D, \emptyset)$, which will prove that $D \rightarrowtail \mathfrak{A}_2(D)$ for all D. However, it is readily observed that $\mathfrak{A}_1$ and $\mathfrak{A}_2$ can be combined in one pass simply by calling $T(D, FA(D))$. In other words, applying the composition of $\mathfrak{A}_1$ with $\mathfrak{A}_2$ to some D produces the same result as $T(D, FA(D))$:

$$\mathfrak{A}_1 \cdot \mathfrak{A}_2 = \lambda\, D\,.\, T(D, FA(D)).$$

Accordingly, we define an algorithm $\mathfrak{A}$ as $\mathfrak{A}(D) = T(D, FA(D))$. In implementation practice, instead of first calling $\mathfrak{A}_2$, then $\mathfrak{MS}$, and then $\mathfrak{A}_1$, as prescribed by 1.4, we can simply call $\mathfrak{A}$ once. (For exposition purposes, we choose to keep the presentations of $\mathfrak{A}_1$ and $\mathfrak{A}_2$ distinct.) The following lemma will prove useful in showing the soundness of $\mathfrak{A}$.

Lemma 1.35 *If $\beta \vdash D \rightsquigarrow Q$ then $\beta \vdash T(D, \beta) \rightsquigarrow Q$.*

Proof: By induction on the structure of D. When D atomic the result is immediate. Let D be a conditional deduction of the form **assume** P **in** D_b, and suppose that $\beta \vdash D \rightsquigarrow P \Rightarrow P'$, so that

$$\beta \cup \{P\} \vdash D_b \rightsquigarrow P'. \tag{1.72}$$

Hence, in this case need to show

$$\beta \vdash T(D, \beta) \rightsquigarrow P \Rightarrow P'. \tag{1.73}$$

From 1.72 and the inductive hypothesis we obtain

$$\beta \cup \{P\} \vdash T(D_b, \beta \cup \{P\}) \rightsquigarrow P'. \tag{1.74}$$

We now distinguish two cases:

$P \in \beta$**:** Then $\beta \cup \{P\} = \beta$, so 1.74 becomes $\beta \vdash T(D_b, \beta) \rightsquigarrow P'$. Accordingly, by the semantics of **cond** (see the Appendix) and compositions, we get

$$\beta \vdash T(D_b, \beta); \textbf{cond } P, \mathcal{C}(T(D_b, \beta)) \rightsquigarrow P \Rightarrow P'. \tag{1.75}$$

But $P \in \beta$ means that

$$T(D, \beta) = T(D_b, \beta); \textbf{cond } P, \mathcal{C}(T(D_b, \beta))$$

so the goal 1.73 follows directly from 1.75.

$P \notin \beta$**:** In this case $T(D, \beta) =$ **assume** P **in** $T(D_b, \beta \cup \{P\})$, so by the semantics of **assume**, 1.73 will follow if we show $\beta \cup \{P\} \vdash T(D_b, \beta \cup \{P\}) \rightsquigarrow P'$. But this is already given by 1.74.

Similar reasoning is used for proofs by contradiction. Finally, supposing that D is of the form $D_1; D_2$, the assumption $\beta \vdash D \rightsquigarrow Q$ means that $\beta \vdash D_1 \rightsquigarrow P_1$ and $\beta \cup \{P_1\} \vdash D_2 \rightsquigarrow Q$, where, of course, $P_1 = \mathcal{C}(D_1)$. Inductively, we get $\beta \vdash T(D_1, \beta) \rightsquigarrow P_1$ and $\beta \cup \{P_1\} \vdash T(D_2, \beta \cup \{P_1\}) \rightsquigarrow Q$. Accordingly,

$$\beta \vdash T(D_1, \beta); T(D_2, \beta \cup \{\mathcal{C}(D_1)\}) \rightsquigarrow Q$$

which is to say, $\beta \vdash T(D) \rightsquigarrow Q$. This completes the inductive argument. ■

Theorem 1.36 *$\mathfrak{A}$ terminates; $D \rightarrowtail \mathfrak{A}(D)$; and $SZ(\mathfrak{A}(D)) \leq SZ(D)$.*

Proof: Termination is obvious. That the size of $\mathfrak{A}(D)$ is never more than the size of D also follows by a straightforward induction on D. Finally, to prove $D \rightarrowtail \mathfrak{A}(D)$, suppose that $\beta \vdash D \rightsquigarrow P$ for some β. By Theorem 1.4, we must have

$$\beta \supseteq FA(D). \tag{1.76}$$

By the same result, $FA(D) \vdash D \rightsquigarrow P$, hence, by Lemma 1.35, $FA(D) \vdash T(D, FA(D)) \rightsquigarrow P$, i.e.,

$$FA(D) \vdash \mathfrak{A}(D) \rightsquigarrow P.$$

Therefore, by 1.76 and dilution we get $\beta \vdash \mathfrak{A}(D) \rightsquigarrow P$, which shows that $D \rightarrowtail \mathfrak{A}(D)$. ■

The final transformation, $\mathfrak{A}_3$, determines whether the hypothesis P of a conditional deduction $D' = \textbf{assume } P \textbf{ in } D_b$ is deduced at a later point, or, more precisely, whether it is deduced somewhere within a deduction dominated by D', as in the following picture:

$$\begin{array}{cc} \vdots & \\ D' = \textbf{assume } P \textbf{ in } D_b; & \\ \vdots & \\ D''; & \qquad \text{(Deduces } P\text{)} \\ \vdots & \end{array}$$

This can lead to the following variant of the detour we discussed earlier:

```
(1) **assume** $\neg\neg A$ **in**
      **begin**
        **dn** $\neg\neg A$;
        **both** $A, B$
      **end**;
(2) **left-and** $\neg\neg A \wedge C$;
(3) **modus-ponens** $\neg\neg A \Rightarrow A \wedge B, \neg\neg A$
```

However, unlike the cases discussed in connection with $\mathfrak{A}_2$ and $\mathfrak{A}_1$, here we cannot hoist the body of (1) above the **assume** (and replace the **assume** by an application of **cond**), because the said body strictly uses the hypothesis $\neg\neg A$, which is neither a free assumption of the overall deduction nor is it deduced *prior* to its hypothetical postulation in (1). Rather, $\neg\neg A$ is deduced *after* the conditional deduction where it appears as a hypothesis. What we will do instead is reduce this case to one that can be handled by the simple hoisting method of algorithm $\mathfrak{A}$. We can do that by "bubbling up" the deduction which derives the hypothesis in question until it precedes the hypothetical deduction, at which point $\mathfrak{A}$ will be able to perform as usual. Specifically, we define:

```
1. $\mathfrak{A}_3(\textbf{assume } P \textbf{ in } D_b) = \textbf{assume } P \textbf{ in } \mathfrak{A}_3(D_b)$
2. $\mathfrak{A}_3(\textbf{suppose-absurd } P \textbf{ in } D_b) = \textbf{suppose-absurd } P \textbf{ in } \mathfrak{A}_3(D_b)$
3. $\mathfrak{A}_3((\textbf{assume } P \textbf{ in } D_b); D) =$
4.     *let* $(D'_b, D') = (\mathfrak{A}_3(D_b), \mathfrak{A}_3(D))$
5.           $(D'', \_, \Delta) = H(D', \{P \Rightarrow \mathcal{C}(D'_b)\})$
6.     *in*
7.       $\overline{\Delta}; \textbf{assume } P \textbf{ in } D'_b; D''$
8. $\mathfrak{A}_3((\textbf{suppose-absurd } P \textbf{ in } D_b); D) =$
9.     *let* $(D'_b, D') = (\mathfrak{A}_3(D_b), \mathfrak{A}_3(D))$
```

```
10.        $(D'', \_, \Delta) = H(D', \{\neg P\})$
11.    *in*
12.        $\overline{\Delta}$; **suppose-absurd** $P$ **in** $D'_b; D''$
13. $\mathfrak{A}_3(D_1; D_2) = \mathfrak{A}_3(D_1); \mathfrak{A}_3(D_2)$
14. $\mathfrak{A}_3(D) = D$
```

Applying this algorith to the deduction above yields:

```
left-and $\neg\neg A \wedge C$;
assume $\neg\neg A$ in
  begin
    dn $\neg\neg A$;
    both $A, B$
  end;
claim $\neg\neg A$;
modus-ponens $\neg\neg A \Rightarrow A \wedge B, \neg\neg A$
```

which will be readily handled by $\mathfrak{A}$. In particular, after applying $\mathfrak{A}$ to the above deduction, followed by $\mathfrak{MS}$ and *contract*, we obtain the final result:

```
left-and $\neg\neg A \wedge C$;
dn $\neg\neg A$;
both $A, B$
```

We can prove:

Theorem 1.37 *$\mathfrak{A}_3$ terminates. Moreover, if D is right-linear then $D \rightarrowtail \mathfrak{A}_3(D)$.*

Proof: Termination is straightforward. We will prove $D \rightarrowtail \mathfrak{A}_3(D)$ by induction on the structure of D. When D is an atomic deduction, the result is immediate. When D is a hypothetical deduction, the result follows by straightforward applications of the inductive hypothesis and Lemma 1.8. Finally, suppose that D is of the form $D_1; D_2$. We distinguish three subscases:

(a) D_1 is of the form **assume** P **in** D_b**:** In this case, letting $D'_b = \mathfrak{A}_3(D_b)$ and $D'_2 = \mathfrak{A}_3(D_2)$, we have

$$\mathfrak{A}_3(D) = \overline{\Delta}; \textbf{assume } P \textbf{ in } D'_b; D''_2 \tag{1.77}$$

where $(D''_2, \_, \Delta) = H(D'_2, \{P \Rightarrow \mathcal{C}(D'_b)\})$. Inductively, $D_b \rightarrowtail D'_b$ and $D_2 \rightarrowtail D'_2$, so, by Lemma 1.8,

$$(\textbf{assume } P \textbf{ in } D_b); D_2 \rightarrowtail (\textbf{assume } P \textbf{ in } D'_b); D'_2. \tag{1.78}$$

Further, Lemma 1.30 implies

$$D'_2 \rightarrowtail \overline{\Delta}; D''_2. \tag{1.79}$$

From 1.79 and 1.78 we get

$$(\textbf{assume } P \textbf{ in } D_b); D_2 \rightarrowtail (\textbf{assume } P \textbf{ in } D'_b); \overline{\Delta}; D''_2. \tag{1.80}$$

By Lemma 1.25, we have $FA(D_x) \cap \{P \Rightarrow \mathcal{C}(D'_b)\} = \emptyset$ for all $D_x \in \Delta$, hence, by Lemma 1.29,

$$(\textbf{assume } P \textbf{ in } D'_b); \overline{\Delta}; D''_2 \rightarrowtail \overline{\Delta}; (\textbf{assume } P \textbf{ in } D'_b); D''_2. \tag{1.81}$$

Finally, from 1.80, 1.81, and the transitivity of $\rightarrowtail$, and in view of 1.77, we conclude $D \rightarrowtail \mathfrak{A}_3(D)$.

(b) D_1 is of the form **suppose-absurd** P **in** D_b**:** The reasoning here is the same as in (a).

(c) None of the above**:** In this case the result follows directly from the inductive hypotheses.

This completes the case analysis and the induction. ■

Finally, we address the question of size—whether $\mathfrak{A}_3(D)$ is always smaller than D. This will usually be the case, but there is an exception similar to that which we discussed in connection with *Hoist*: when algorithm H inserts a tail-position claim in D'' (lines 5 and 10). This will increase the size of the resulting deduction by one. (any other claims generated by H will be eliminated later by the claim-removal algorithm, $\mathfrak{C}$, as guaranteed by Lemma 1.16). But it is easy to avoid this special case, since H is defined so that whenever a tail-position claim is appended to D'', the last deduction of the list Δ has the same conclusion as the proposition asserted by the said claim. But if this is the case we can do away with D'' altogether, as well with the **assume** P **in** D'_b, and simply output Δ (and likewise for the **suppose-absurd**), in which case the size of the resulting deduction will be *strictly* smaller than that of the original. Accordingly, we modify lines 7 and 12 to be as follows, respectively:

$$\Delta \neq [] \ \ \mathit{and} \ \ \mathcal{C}(\overline{\Delta}) = \mathcal{C}(D'') \,?\, \rightarrow \overline{\Delta},\ \overline{\Delta};\underline{\textbf{assume}}\ P\ \textbf{in}\ D'_b; D''$$

and

$$\Delta \neq [] \ \ \mathit{and} \ \ \mathcal{C}(\overline{\Delta}) = \mathcal{C}(D'') \,?\, \rightarrow \overline{\Delta},\ \overline{\Delta};\underline{\textbf{suppose-absurd}}\ P\ \textbf{in}\ D'_b; D''.$$

This affects neither termination nor the property $D \rightarrowtail \mathfrak{A}_3(D)$ (on the assumption that D is right-linear), since the reduction is performed only if $\mathcal{C}(\overline{\Delta}) = \mathcal{C}(D'')$, so Theorem 1.37 continues to hold. Further, the modification guarantees that every claim inserted by H will eventually be removed by $\mathfrak{C}$, which ensures that the ultimate result of the simplification procedure will never be of greater size than the original.[3]

In conclusion, we define

$$\mathit{restructure} \ = \ \mathfrak{MS} \cdot \mathfrak{A} \cdot \mathfrak{MS} \cdot \mathfrak{A}_3 \cdot \mathfrak{MS}$$

and

$$\mathit{simplify} \ = \ \mathit{contract} \cdot \mathit{restructure}.$$

Putting together the various preceding results will show that *simplify* always terminates and that $D \rightarrowtail \mathit{simplify}(D)$. Size is always either strictly decreased or preserved, except by *Hoist*, during the application of $\mathfrak{MS}$, and by $\mathfrak{A}_3$. Both of these transformations may introduce some additional trivial claims. However, we have taken care to define $\mathfrak{MS}$ and $\mathfrak{A}_3$ so that all such claims will be in non-tail positions and will thus be eventually eliminated by the claim-removal algorithm, $\mathfrak{C}$. Therefore, we conclude:

Theorem 1.38 *simplify always terminates;* $D \rightarrowtail \mathit{simplify}(D)$*;* $SZ(\mathit{simplify}(D)) \leq SZ(D)$.

1.5 Examples

In this section we illustrate *simplify* with some simple examples of the "detours" shown in Figure 1.4. We will write **mp** and **dn** as abbreviations for **modus-ponens** and **double-negation**, respectively. We begin with a couple of examples of detour (c):

[3]Moreover, to avoid gratuitous hoistings, in practice we will perform these restructurings only if the hypothesis P is in fact derived within D (lines 3 and 8). See the implementation of $\mathfrak{A}_3$ in Figure 1.11.

$$\mathrm{D} = \begin{array}{l} \textbf{dn}\ \neg\neg A; \\ \textbf{assume}\ A\ \textbf{in} \\ \quad \textbf{both}\ A, B; \\ \textbf{mp}\ A \Rightarrow A \wedge B, A \end{array} \xrightarrow{\textit{restructure}} \begin{array}{l} \textbf{dn}\ \neg\neg A; \\ \textbf{both}\ A, B; \\ \textbf{cond}\ A, A \wedge B; \\ \textbf{mp}\ A \Rightarrow A \wedge B, A \end{array} \xrightarrow{\textit{contract}} \begin{array}{l} \textbf{dn}\ \neg\neg A; \\ \textbf{both}\ A, B; \end{array}$$

An alternative form of the same detour is obtained by swapping the order of **dn** and **assume** in the above deduction. We see that *simplify* handles this with the same ease:

$$D = \begin{array}{l} \textbf{assume}\ A\ \textbf{in} \\ \quad \textbf{both}\ A, B; \\ \textbf{dn}\ \neg\neg A; \\ \textbf{mp}\ A \Rightarrow A \wedge B, A \end{array} \xrightarrow{\textit{restructure}} \begin{array}{l} \textbf{dn}\ \neg\neg A; \\ \textbf{both}\ A, B; \\ \textbf{cond}\ A, A \wedge B; \\ \textbf{claim}\ A; \\ \textbf{mp}\ A \Rightarrow A \wedge B, A \end{array} \xrightarrow{\textit{contract}} \begin{array}{l} \textbf{dn}\ \neg\neg A; \\ \textbf{both}\ A, B; \end{array}$$

We continue with detour (a), based on negation:

$$D = \begin{array}{l} \textbf{left-and}\ A \wedge B; \\ \textbf{suppose-absurd}\ \neg A\ \textbf{in} \\ \quad \textbf{absurd}\ A, \neg A; \\ \textbf{dn}\ \neg\neg A \end{array} \xrightarrow{\textit{restructure}} \begin{array}{l} \textbf{left-and}\ A \wedge B; \\ \textbf{suppose-absurd}\ \neg A\ \textbf{in} \\ \quad \textbf{absurd}\ A, \neg A; \\ \textbf{dn}\ \neg\neg A \end{array} \xrightarrow{\textit{contract}} \textbf{left-and}\ A \wedge B$$

A slightly trickier variant of the same detour is shown in the next deduction:

$$\mathrm{D} = \begin{array}{l} \textbf{suppose-absurd}\ \neg A\ \textbf{in} \\ \ \textbf{begin} \\ \quad \textbf{left-and}\ A \wedge B; \\ \quad \textbf{absurd}\ A, \neg A \\ \ \textbf{end}; \\ \textbf{dn}\ \neg\neg A \end{array}$$

Here *simplify* will operate as follows:

$$\begin{array}{l} \textbf{suppose-absurd}\ \neg A\ \textbf{in} \\ \ \textbf{begin} \\ \quad \textbf{left-and}\ A \wedge B; \\ \quad \textbf{absurd}\ A, \neg A \\ \ \textbf{end}; \\ \textbf{dn}\ \neg\neg A \end{array} \xrightarrow{\textit{restructure}} \begin{array}{l} \textbf{left-and}\ A \wedge B; \\ \textbf{claim}\ A; \\ \textbf{claim}\ A; \\ \textbf{suppose-absurd}\ \neg A\ \textbf{in} \\ \quad \textbf{begin} \\ \qquad \textbf{claim}\ A; \\ \qquad \textbf{absurd}\ A, \neg A \\ \quad \textbf{end}; \\ \textbf{dn}\ \neg\neg A \end{array} \xrightarrow{\textit{contract}} \textbf{left-and}\ A \wedge B$$

Next we illustrate a disjunction detour. Let D be the following deduction:

dn $\neg\neg(A_1 \wedge B)$;
left-either $A_1 \wedge B, A_2 \wedge B$;
assume $A_1 \wedge B$ **in**
  right-and $A_1 \wedge B$;
assume $A_2 \wedge B$ **in**
  right-and $A_2 \wedge B$;
cd $(A_1 \wedge B) \vee (A_2 \wedge B), (A_1 \wedge B) \Rightarrow B, (A_2 \wedge B) \Rightarrow B$

We have:

$$D \xrightarrow{\textit{restructure}} \begin{array}{l} \textbf{dn}\ \neg\neg(A_1 \wedge B); \\ \textbf{left-either}\ A_1 \wedge B, A_2 \wedge B; \\ \textbf{right-and}\ A_1 \wedge B; \\ \textbf{cond}\ A_1 \wedge B, B; \\ \textbf{assume}\ A_2 \wedge B\ \textbf{in} \\ \quad \textbf{right-and}\ A_2 \wedge B; \\ \textbf{cd}\ (A_1 \wedge B) \vee (A_2 \wedge B), (A_1 \wedge B) \Rightarrow B, (A_2 \wedge B) \Rightarrow B \end{array} \xrightarrow{\textit{contract}} \begin{array}{l} \textbf{dn}\ \neg\neg(A_1 \wedge B); \\ \textbf{right-and}\ A_1 \wedge B \end{array}$$

A variant of this detour is contained in the following deduction, call it D:

assume $A_1 \wedge B$ **in**
  right-and $A_1 \wedge B$;
assume $A_2 \wedge B$ **in**
  right-and $A_2 \wedge B$;
dn $\neg\neg(A_2 \wedge B)$;
right-either $A_1 \wedge B, A_2 \wedge B$;
cd $(A_1 \wedge B) \vee (A_2 \wedge B), (A_1 \wedge B) \Rightarrow B, (A_2 \wedge B) \Rightarrow B$

In this case we have:

$$D \xrightarrow{\textit{restructure}} \begin{array}{l} \textbf{assume}\ A_1 \wedge B\ \textbf{in} \\ \quad \textbf{right-and}\ A_1 \wedge B; \\ \textbf{dn}\ \neg\neg(A_2 \wedge B); \\ \textbf{right-either}\ A_1 \wedge B, A_2 \wedge B; \\ \textbf{right-and}\ A_2 \wedge B; \\ \textbf{cond}\ A_2 \wedge B, B; \\ \textbf{claim}\ A_2 \wedge B; \\ \textbf{claim}\ (A_1 \wedge B) \vee (A_2 \wedge B); \\ \textbf{cd}\ (A_1 \wedge B) \vee (A_2 \wedge B), (A_1 \wedge B) \Rightarrow B, (A_2 \wedge B) \Rightarrow B \end{array} \xrightarrow{\textit{contract}} \begin{array}{l} \textbf{dn}\ \neg\neg(A_2 \wedge B); \\ \textbf{right-and}\ A_2 \wedge B \end{array}$$

We close with a biconditional detour. Let D be the following deduction:

assume $A \wedge B$ **in**
 begin
  left-and $A \wedge B$;
  right-and $A \wedge B$;
  both B, A

```
  end;
assume $B \wedge A$ in
  begin
    right-and $B \wedge A$;
    left-and $B \wedge A$;
    both $A, B$
  end;
equivalence $A \wedge B \Rightarrow B \wedge A, B \wedge A \Rightarrow A \wedge B$;
left-iff $A \wedge B \Leftrightarrow B \wedge A$
```

We have:

$$D \xrightarrow{\textit{restructure}} D \xrightarrow{\textit{contract}} \begin{array}{l}\textbf{assume } A \wedge B \textbf{ in}\\ \quad\textbf{begin}\\ \quad\quad\textbf{left-and } A \wedge B;\\ \quad\quad\textbf{right-and } A \wedge B;\\ \quad\quad\textbf{both } B, A\\ \quad\textbf{end}\end{array}$$

1.6 Implementation

In this section we present SML-NJ code implementing every transformation presented in this paper.[4] Figure 1.7 depicts SML-NJ datatypes encoding the abstract syntax of propositions and proofs, along with some auxiliary functions. Note that there is a bit (a `bool` field) associated with every hypothetical deduction. This bit is used to indicate whether or not the hypothetical deduction is "marked", as required by the hoisting algorithm discussed in Section 1.4. When a proof is initially constructed, all such bits should be `false` to signify that the corresponding hypothetical deductions have not been processed yet.

Implementations of $\mathcal{C}$ and *FA* appear in Figure 1.8. Here *FA* returns a list of propositions, rather than a set. We take care to remove duplicates from such lists; this can lighten the load of some transformations. The computation of a proof's conclusion will succeed only if the proof is well-formed, as prescribed by the relevant inference system in the Appendix. An exception `IllFormedProof` will be raised otherwise. The contraction procedures are implemented in Figure 1.9, and the scope-maximization algorithm in Figure 1.10. Finally, the global transformations $\mathfrak{A}$ and $\mathfrak{A}_3$ are shown in Figure 1.11. Observe that the $\mathfrak{A}_3$ restructuring of a hypothetical deduction (**assume** P **in** D_b); D (or (**suppose-absurd** P **in** D_b); D) is carried out only if we are certain that the hypothesis P is deduced inside D, which we determine by checking `if memberOf (map concl (getThreadElements D)) P`. This is done to avoid disrupting the structure of the original deduction without reason.

Finally, we note that the two major contraction procedures introduced in this paper, the productivity and parsimony analyses, are applicable to any type-α DPL [1] that features a cut operator (such as ";") and for which the two functions $\mathcal{C}(D)$ and $FA(D)$ are definable for any given D in a way that preserves their meaning in $\mathcal{NDL}$, i.e., so that

1. $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$; and
2. $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ iff $\beta \supseteq FA(D)$

[4]An online copy of this code can be found at `www.ai.mit.edu/projects/dynlangs/dpls/alpha-simp.sml`.

for all D and β. As long as the proof theory of the DPL is as described in Section 1.2, the transformations will be safe in the expected sense. Since this is the case for most type-α DPLs, these procedures could be applied with minimum modification to any such language. We leave it as an exercise for the reader to recast the given algorithms at a more abstract level by using an SML-NJ functor parameterized over a structure with types such as `prop` and `proof` and functions such as `is-composition:proof -> bool`, `concl:proof -> prop`, and `fa:  proof -> prop list`.

```
structure Simplify =
struct

datatype prim_rule = claim | dn | mp | both | leftAnd | rightAnd | cd
                   | leftEither | rightEither | equiv | leftIff | rightIff
                   | absurd | trueIntro | condRule | negRule;

datatype prop =  atom of string
               | trueProp
               | falseProp
               | neg of prop
               | conj of prop * prop
               | disj of prop * prop
               | cond of prop * prop
               | biCond of prop * prop;

datatype proof = ruleApp of prim_rule * prop list
               | assumeProof of prop * proof * bool
               | supAbProof of prop * proof * bool
               | compProof of proof * proof;


exception IllFormedProof;

fun illFormed() = raise IllFormedProof;

fun fp f = fn D => let val D' = f D
                    in
                       if D = D' then D else (fp f) D'
                    end;

fun weave f [] = f
  | weave f (g::rest) = f o g o (weave f rest);

fun memberOf L x = List.exists (fn a => a = x) L;

fun emptyIntersection(L1,L2) = not (List.exists (memberOf L2) L1);

fun remove(x,L) = List.filter (fn y => not(x = y)) L;

fun removeDuplicates [] = []
  | removeDuplicates (x::rest) = x::removeDuplicates(remove(x,rest));

fun getThreadElements(compProof(D1,D2)) = D1::getThreadElements(D2)
  | getThreadElements(D) = [D];

fun makeThread([D]) = D
  | makeThread(D::rest) = compProof(D,makeThread(rest))

fun isClaim(ruleApp(claim,_)) = true
  | isClaim(_) = false
```

Figure 1.7: Abstract syntax and some auxiliary functions.

```
fun ruleConcl(claim,[P]) = P
  | ruleConcl(dn,[neg(neg(P))]) = P
  | ruleConcl(condRule,[P,Q]) = cond(P,Q)
  | ruleConcl(negRule,[P]) = neg(P)
  | ruleConcl(mp,[cond(P1,P2),P3]) = if (P1 = P3) then P2 else illFormed()
  | ruleConcl(both,[P1,P2]) = conj(P1,P2)
  | ruleConcl(leftAnd,[conj(P1,P2)]) = P1
  | ruleConcl(rightAnd,[conj(P1,P2)]) = P2
  | ruleConcl(equiv,[cond(P1,P2),cond(P3,P4)]) =
      if P1 = P4 andalso P2 = P3 then biCond(P1,P2) else illFormed()
  | ruleConcl(leftIff,[biCond(P,Q)]) = cond(P,Q)
  | ruleConcl(rightIff,[biCond(P,Q)]) = cond(Q,P)
  | ruleConcl(leftEither,[P,Q]) = disj(P,Q)
  | ruleConcl(rightEither,[P,Q]) = disj(P,Q)
  | ruleConcl(cd,[disj(P1,P2),cond(P3,Q),cond(P4,Q')]) =
     if P1 = P3 andalso P2 = P4 andalso Q = Q' then Q else illFormed()
  | ruleConcl(absurd,[P1,neg(P2)]) = if P1 = P2 then falseProp else illFormed()
  | ruleConcl(trueIntro,[]) = trueProp
  | ruleConcl(_) = illFormed();

fun concl(ruleApp(M,args)) = ruleConcl(M,args)
  | concl(assumeProof(P,D,_)) = cond(P,concl(D))
  | concl(supAbProof(P,D,_)) = neg(P)
  | concl(compProof(_,D2)) = concl(D2);

fun fa D =
 let fun h(ruleApp(leftEither,[P1,P2])) = [P1]
       | h(ruleApp(rightEither,[P1,P2])) = [P2]
       | h(ruleApp(M,args)) = args
       | h(assumeProof(P,D,_)) = remove(P,h(D))
       | h(supAbProof(P,D,_)) = remove(P,h(D))
       | h(compProof(D1,D2)) =
           h(D1)@(remove(concl(D1),h(D2)))
 in
   removeDuplicates(h(D))
 end;
```

Figure 1.8: Computing conclusions and free assumptions.

```
fun makeStrict(assumeProof(P,D,mark)) = assumeProof(P,makeStrict(D),mark)
  | makeStrict(supAbProof(P,D,mark)) = supAbProof(P,makeStrict(D),mark)
  | makeStrict(compProof(D1,D2)) =
     let val D1' = makeStrict(D1)
         val D2' = makeStrict(D2)
     in
        if memberOf (fa D2') (concl D1') then compProof(D1',D2') else D2'
     end
  | makeStrict(D) = D;

fun removeRepetitions(D) =
 let fun RR(D,L) =
        let val P = concl(D)
        in
           if memberOf L P then ruleApp(claim,[P])
           else
              case D of
                 assumeProof(hyp,D_b,mark) => assumeProof(hyp,RR(D_b,hyp::L),mark)
               | supAbProof(hyp,D_b,mark) => supAbProof(hyp,RR(D_b,hyp::L),mark)
               | compProof(D1,D2) => let val D1' = RR(D1,L)
                                     in
                                        compProof(D1',RR(D2,concl(D1')::L))
                                     end
               | _ => D
        end
 in
    RR(D,[])
 end;

fun elimClaims(assumeProof(P,D_b,mark)) = assumeProof(P,elimClaims(D_b),mark)
  | elimClaims(supAbProof(P,D_b,mark)) = supAbProof(P,elimClaims(D_b),mark)
  | elimClaims(compProof(D1,D2)) =
      let val (D1',D2') = (elimClaims(D1),elimClaims(D2))
          val comp = compProof(D1',D2')
      in
         (case D1' of
             ruleApp(claim,_) => D2'
           | _ => (case D2' of
                     ruleApp(claim,_) =>
                       if concl(D1') = concl(D2') then D1' else comp
                   | _ => comp))
      end
  | elimClaims(D) = D;

val contract = fp (elimClaims o removeRepetitions o makeStrict);
```

Figure 1.9: The contraction algorithms.

```
fun H(compProof(D1,D2),L) =
             let val (D1',L1,Delta1) = H(D1,L)
                 val (D2',L2,Delta2) = H(D2,L1)
             in
                (compProof(D1',D2'),L2,Delta1 @ Delta2)
             end
  | H(D,L) = let val C = concl(D)
             in
               if emptyIntersection(fa(D),L) then
                  (ruleApp(claim,[C]),L,[D])
               else
                  (D,C::L,[])
             end;

val maximizeScope =
 let val rightLinearize =
           let fun rl(assumeProof(P,D,b)) = assumeProof(P,rl(D),b)
                 | rl(supAbProof(P,D,b)) = supAbProof(P,rl(D),b)
                 | rl(compProof(D_l,D_r)) =
                      (case D_l of
                          compProof(D1,D2) => rl(compProof(D1,compProof(D2,D_r)))
                        | _ => compProof(rl(D_l),rl(D_r)))
                 | rl(D) = D
           in
             rl
           end
     fun allMarked(assumeProof(_,D,mark)) = mark andalso allMarked(D)
       | allMarked(supAbProof(_,D,mark)) =  mark andalso allMarked(D)
       | allMarked(compProof(D1,D2)) = allMarked(D1) andalso allMarked(D2)
       | allMarked(ruleApp(_)) = true
     fun hoist(assumeProof(P,D_b,mark as false)) =
           if allMarked(D_b) then
              let val (D_b',_,Delta) = H(D_b,[P])
              in
                if not(null(Delta)) andalso concl(makeThread(Delta)) = concl(D_b')
                then makeThread(Delta@[ruleApp(condRule,[P,concl(D_b')])])
                else
                   makeThread(Delta@[assumeProof(P,D_b',true)])
              end
           else
              assumeProof(P,hoist(D_b),mark)
       | hoist(supAbProof(P,D_b,mark as false)) =
           if allMarked(D_b) then
              let val (D_b',_,Delta) = H(D_b,[P])
              in
                if not(null(Delta)) andalso concl(makeThread(Delta)) = concl(D_b')
                then makeThread(Delta@[ruleApp(negRule,[P])])
                else
                   makeThread(Delta@[supAbProof(P,D_b',true)])
              end
           else
              supAbProof(P,hoist(D_b),mark)
       | hoist(compProof(D1,D2)) = compProof(hoist(D1),hoist(D2))
       | hoist(D) = D
 in
    (fp (rightLinearize o hoist)) o rightLinearize
 end;
```

Figure 1.10: Scope-maximization algorithms.

```
fun A(D) =
  let fun T(assumeProof(P,D_b,mark),L) =
          if memberOf L P then
             let val D_b' = T(D_b,L)
             in
                compProof(D_b',ruleApp(condRule,[P,concl(D_b')]))
             end
          else
             assumeProof(P,T(D_b,P::L),mark)
        | T(supAbProof(P,D_b,mark),L) =
          if memberOf L P then
             let val D_b' = T(D_b,L)
             in
               compProof(D_b',ruleApp(negRule,[P]))
             end
          else
             supAbProof(P,T(D_b,P::L),mark)
        | T(compProof(D1,D2),L) =
            let val D1' = T(D1,L)
            in
              compProof(D1',T(D2,concl(D1')::L))
            end
        | T(D,_) = D
  in
     T(D,fa(D))
  end;

fun A3(assumeProof(P,D,mark)) = assumeProof(P,A3(D),mark)
  | A3(supAbProof(P,D,mark)) = supAbProof(P,A3(D),mark)
  | A3(compProof(assumeProof(P,D_b,mark),D)) =
        let val (D_b',D') = (A3(D_b),A3(D))
            val (D'',_,Delta) = H(D',[cond(P,concl(D_b'))])
        in
          if memberOf (map concl (getThreadElements D)) P
          then
             (if not(null(Delta)) andalso concl(makeThread(Delta)) = concl(D'')
              then makeThread(Delta)
              else
                 makeThread(Delta@[assumeProof(P,D_b',mark),D''])) 
          else
              compProof(assumeProof(P,D_b',mark),D')
        end
  | A3(compProof(supAbProof(P,D_b,mark),D)) =
        let val (D_b',D') = (A3(D_b),A3(D))
            val (D'',_,Delta) = H(D',[neg(P)])
        in
           if memberOf (map concl (getThreadElements D)) P
           then
               (if not(null(Delta)) andalso concl(makeThread(Delta)) = concl(D'')
                then makeThread(Delta)
                else
                   makeThread(Delta@[supAbProof(P,D_b',mark),D''])) 
           else
               compProof(supAbProof(P,D_b',mark),D')
        end
  | A3(compProof(D1,D2)) = compProof(A3(D1),A3(D2))
  | A3(D) = D;

val restructure =  weave maximizeScope [A,A3];
val simplify = contract o restructure;
end; (* of structure Simplify *)
```

Figure 1.11: Global transformations.

Appendix

In this appendix we give rigorous definitions of some $\mathcal{NDL}$ notions that appear in the body of the paper.

The *domain* of a deduction D, written $Dom(D)$, is a set of integer lists defined as follows:

$$\begin{array}{rcl}
Dom(Rule\ P_1,\ldots,P_n) & = & \{\,[]\} \cup \{\,[1],\ldots,[n]\} \\
Dom(\textbf{assume}\ P\ \textbf{in}\ D) & = & \{\,[]\} \cup \{\,[1]\} \cup \{\,2{::}p \mid p \in Dom(D)\} \\
Dom(\textbf{suppose-absurd}\ P\ \textbf{in}\ D) & = & \{\,[]\} \cup \{\,[1]\} \cup \{\,2{::}p \mid p \in Dom(D)\} \\
Dom(D_1; D_2) & = & \{\,[]\} \cup \{\,1{::}p \mid p \in Dom(D_1)\} \cup \{\,2{::}p \mid p \in Dom(D_2)\}
\end{array}$$

Next, we define a function *Label* that takes a deduction D and a "position" $p \in Dom(D)$ and returns whatever part of D appears there:

$$\begin{array}{rcl}
Label(P,[]) & = & P \\
Label(Prim\text{-}Rule\ P_1,\ldots,P_n,[]) & = & Prim\text{-}Rule \\
Label(Prim\text{-}Rule\ P_1,\ldots,P_n,[i]) & = & P_i\ (\text{for } i=1,\ldots,n) \\
Label(\textbf{assume}\ P\ \textbf{in}\ D,[]) & = & \textbf{assume} \\
Label(\textbf{assume}\ P\ \textbf{in}\ D,[1]) & = & P \\
Label(\textbf{assume}\ P\ \textbf{in}\ D,2{::}p) & = & Label(D,p) \\
Label(\textbf{suppose-absurd}\ P\ \textbf{in}\ D,[]) & = & \textbf{suppose-absurd} \\
Label(\textbf{suppose-absurd}\ P\ \textbf{in}\ D,[1]) & = & P \\
Label(\textbf{suppose-absurd}\ P\ \textbf{in}\ D,2{::}p) & = & Label(D,p) \\
Label(D_1;D_2,[]) & = & ; \\
Label(D_1;D_2,1{::}p) & = & Label(D_1,p) \\
Label(D_1;D_2,2{::}p) & = & Label(D_2,p)
\end{array}$$

We say that a deduction D' *occurs in* D *at* some $p \in Dom(D)$ iff $Label(D',q) = Label(D,p \oplus q)$ for every $q \in Dom(D')$. By a *subdeduction* of D we will mean any deduction that occurs in D at some position. We define a *thread* of D as any subdeduction of D of the form $D_1; D_2; \ldots; D_n; D_{n+1}$, $n \geq 1$.[5] (Occasionally it is also convenient to view a deduction D that is not a composition as a single-element thread.) We call $D_1,\ldots,D_n$ the *elements* of the thread. For each $i = 1,\ldots,n$, we say that D_i *dominates* every D_j, $i < j \leq n+1$; and we call $D_1,\ldots,D_n$ the *dominating elements* of the thread. In general, we say that a deduction D' dominates a deduction D'' in D iff there is a position $p \in Dom(D)$ and a non-empty list $q = [2,\ldots,2]$ (i.e., q is a list of one or more 2s) such that

1. $Label(D,p) = ;$
2. D' occurs in D at $p \oplus [1]$
3. $Label(D,p \oplus q') = ;$ for every prefix $q' \sqsubseteq q$
4. D'' occurs in D at $p \oplus q \oplus [1]$.

Clearly, the dominance relation imposes a total ordering on the elements of a thread.

We write $D' \blacktriangleleft D$ to indicate that D' is a subdeduction of D that occurs *in tail position* inside D. This relation is precisely defined by the following rules:

[5]Or more precisely, recalling that composition associates to the right, as any subdeduction of the form $D_1; (D_2; \cdots; (D_n; D_{n+1}) \cdots)$.

$$\frac{}{D \blacktriangleleft D} \qquad \frac{D' \blacktriangleleft D}{D' \blacktriangleleft \textbf{assume } P \textbf{ in } D}$$

$$\frac{D' \blacktriangleleft D}{D' \blacktriangleleft \textbf{suppose-absurd } P \textbf{ in } D} \qquad \frac{D_2' \blacktriangleleft D_2}{D_2' \blacktriangleleft D_1; D_2}$$

We introduce the notion of a *well-formed deduction* via rules that establish judgements of the form $\vdash_W D$ (read "D is well-formed.") For atomic deductions we have the following axiom schemas:

$$\frac{}{\vdash_W \textbf{both } P, Q} \qquad \frac{}{\vdash_W \textbf{left-and } P \wedge Q} \qquad \frac{}{\vdash_W \textbf{right-and } P \wedge Q}$$

$$\frac{}{\vdash_W \textbf{modus-ponens } P \Rightarrow Q, P} \qquad \frac{}{\vdash_W \textbf{left-either } P, Q} \qquad \frac{}{\vdash_W \textbf{right-either } P, Q}$$

$$\frac{}{\vdash_W \textbf{cd } P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q} \qquad \frac{}{\vdash_W \textbf{cond } P, Q}$$

$$\frac{}{\vdash_W \textbf{equivalence } P \Rightarrow Q, Q \Rightarrow P} \qquad \frac{}{\vdash_W \textbf{absurd } P, \neg P} \qquad \frac{}{\vdash_W \textbf{neg } P}$$

$$\frac{}{\vdash_W \textbf{double-negation } \neg\neg P} \qquad \frac{}{\vdash_W \textbf{left-iff } P \Leftrightarrow Q} \qquad \frac{}{\vdash_W \textbf{right-iff } P \Leftrightarrow Q}$$

For compound deductions we have:

$$\frac{\vdash_W D}{\vdash_W \textbf{assume } P \textbf{ in } D} \qquad \frac{\vdash_W D}{\vdash_W \textbf{suppose-absurd } P \textbf{ in } D} \qquad \frac{\vdash_W D_1 \quad \vdash_W D_2}{\vdash_W D_1; D_2}$$

The *conclusion* $\mathcal{C}(D)$ of a well-formed atomic deduction D is defined as:

$$\begin{array}{rcl}
\mathcal{C}(\textbf{modus-ponens } P \Rightarrow Q, P) & = & Q \\
\mathcal{C}(\textbf{double-negation } \neg\neg P) & = & P \\
\mathcal{C}(\textbf{both } P, Q) & = & P \wedge Q \\
\mathcal{C}(\textbf{left-and } P \wedge Q) & = & P \\
\mathcal{C}(\textbf{right-and } P \wedge Q) & = & Q \\
\mathcal{C}(\textbf{left-either } P, Q) & = & P \vee Q \\
\mathcal{C}(\textbf{right-either } P, Q) & = & P \vee Q \\
\mathcal{C}(\textbf{cd } P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q) & = & Q \\
\mathcal{C}(\textbf{equivalence } P \Rightarrow Q, Q \Rightarrow P) & = & P \Leftrightarrow Q \\
\mathcal{C}(\textbf{left-iff } P \Leftrightarrow Q) & = & P \Rightarrow Q \\
\mathcal{C}(\textbf{right-iff } P \Leftrightarrow Q) & = & Q \Rightarrow P \\
\mathcal{C}(\textbf{neg } P) & = & \neg P \\
\mathcal{C}(\textbf{cond } P, Q) & = & P \Rightarrow Q \\
\mathcal{C}(\textbf{absurd } P, \neg P) & = & \textbf{false}
\end{array}$$

For non-primitive deductions we have:

$$\begin{array}{rcl}
\mathcal{C}(\textbf{assume } P \textbf{ in } D) & = & P \Rightarrow \mathcal{C}(D) \\
\mathcal{C}(\textbf{suppose-absurd } P \textbf{ in } D) & = & \neg P \\
\mathcal{C}(D_1; D_2) & = & \mathcal{C}(D_2)
\end{array}$$

Finally, Figure 1.12 depicts the semantics of primitive rule applications. Observe the specification of **cond** and **neg**. The first of these allows us to introduce a conditional $P \Rightarrow Q$ whenever the conclusion Q is already established (in the assumption base). The second, **neg**, allows us to negate

$$\beta \cup \{P\} \vdash \textbf{claim } P \rightsquigarrow P$$
$$\beta \cup \{P \Rightarrow Q, P\} \vdash \textbf{modus-ponens } P \Rightarrow Q, P \rightsquigarrow Q$$
$$\beta \cup \{\neg\neg P\} \vdash \textbf{double-negation } \neg\neg P \rightsquigarrow P$$
$$\beta \cup \{P_1, P_2\} \vdash \textbf{both } P_1, P_2 \rightsquigarrow P_1 \wedge P_2$$
$$\beta \cup \{P_1 \wedge P_2\} \vdash \textbf{left-and } P_1 \wedge P_2 \rightsquigarrow P_1$$
$$\beta \cup \{P_1 \wedge P_2\} \vdash \textbf{right-and } P_1 \wedge P_2 \rightsquigarrow P_2$$
$$\beta \cup \{P_1\} \vdash \textbf{left-either } P_1, P_2 \rightsquigarrow P_1 \vee P_2$$
$$\beta \cup \{P_2\} \vdash \textbf{right-either } P_1, P_2 \rightsquigarrow P_1 \vee P_2$$
$$\beta \cup \{P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q\} \vdash \textbf{cd } P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q \rightsquigarrow Q$$
$$\beta \cup \{P_1 \Rightarrow P_2, P_2 \Rightarrow P_1\} \vdash \textbf{equivalence } P_1 \Rightarrow P_2, P_2 \Rightarrow P_1 \rightsquigarrow P_1 \Leftrightarrow P_2$$
$$\beta \cup \{P_1 \Leftrightarrow P_2\} \vdash \textbf{left-iff } P_1 \Leftrightarrow P_2 \rightsquigarrow P_1 \Rightarrow P_2$$
$$\beta \cup \{P_1 \Leftrightarrow P_2\} \vdash \textbf{right-iff } P_1 \Leftrightarrow P_2 \rightsquigarrow P_2 \Rightarrow P_1$$
$$\beta \vdash \textbf{true-intro} \rightsquigarrow \textbf{true}$$
$$\beta \cup \{P, \neg P\} \vdash \textbf{absurd } P, \neg P \rightsquigarrow \textbf{false}$$
$$\beta \cup \{Q\} \vdash \textbf{cond } P, Q \rightsquigarrow P \Rightarrow Q$$
$$\beta \cup \{\textbf{false}\} \vdash \textbf{neg } P \rightsquigarrow \neg P$$

Figure 1.12: Evaluation axioms for rule applications.

any proposition P, provided that **false** is in the assumption base. Both rules are obviously sound. Neither is necessary: **cond** is subsumed by **assume**, and **neg** by **suppose-absurd**. They are included here because they allow more succinct deductions whenever the aforementioned conditions obtain. For instance, if we already know that B is in the assumption base, we can introduce $A \Rightarrow B$ with the single rule application **cond** A, B, instead of the slightly larger

$$\textbf{assume } A \textbf{ in claim } B.$$

And likewise, whenever **false** is known to be in the assumption base, we can introduce a negation $\neg P$ with the single application **neg** P instead of

$$\textbf{suppose-absurd } P \textbf{ in claim false}.$$

Bibliography

[1] K. Arkoudas. Type-α DPLs. MIT AI Memo 2001-25.

[2] A. G. Dragalin. *Mathematical Intuitionism. Introduction to Proof Theory*, volume 67 of *Translations of Mathematical Monographs.* American Mathematical Society, Providence, RI, 1988.

[3] G. Gentzen. *The collected papers of Gerhard Gentzen.* North-Holland, Amsterdam, Holland, 1969. English translations of Gentzen's papers, edited and introduced by M. E. Szabo.

[4] D. Prawitz. *Natural Deduction.* Almqvist & Wiksell, Stockhol, Sweden, 1965.