

# Lecture Notes on Constructive Logic: Overview

15-317: Constructive Logic  
Frank Pfenning

Lecture 1  
August 29, 2017

## 1 Introduction

According to Wikipedia, logic is the study of the principles of valid inferences and demonstration. From the breadth of this definition it is immediately clear that logic constitutes an important area in the disciplines of philosophy and mathematics. Logical tools and methods also play an essential role in the design, specification, and verification of computer hardware and software. It is these applications of logic in computer science which will be the focus of this course. In order to gain a proper understanding of logic and its relevance to computer science, we will need to draw heavily on the much older logical traditions in philosophy and mathematics. We will discuss some of the relevant history of logic and pointers to further reading throughout these notes. In this introduction, we give only a brief overview of the goal, contents, and approach of this class.

## 2 Topics

The course is divided into four parts:

- I. Proofs as Evidence for Truth
- II. Proofs as Programs
- III. Proof Search as Computation
- IV. Substructural and Modal Logics

Proofs are central in all parts of the course, and give it its constructive nature. In each part, we will exhibit connections between proofs and forms of computations studied in computer science. These connections will take quite different forms, which shows the richness of logic as a foundational discipline at the nexus between philosophy, mathematics, and computer science.

In Part I we establish the basic vocabulary and systematically study propositions and proofs, mostly from a philosophical perspective. The treatment will be rather formal in order to permit an easy transition into computational applications. We will also discuss some properties of the logical systems we develop and strategies for proof search. We aim at a systematic account for the usual forms of logical expression, providing us with a flexible and thorough foundation for the remainder of the course. We will also highlight the differences between constructive and non-constructive reasoning. Exercises in this section will test basic understanding of logical connectives and how to reason with them.

In Part II we focus on constructive reasoning. This means we consider only proofs that describe algorithms. This turns out to be quite natural in the framework we have established in Part I. In fact, it may be somewhat surprising that many proofs in mathematics today are *not* constructive in this sense. Concretely, we find that for a certain fragment of logic, constructive proofs correspond to functional programs and vice versa. More generally, we can extract functional programs from constructive proofs of their specifications. We often refer to constructive reasoning as *intuitionistic*, while non-constructive reasoning is *classical*. Exercises in this part explore the connections between proofs and programs, and between theorem proving and programming.

In Part III we study a different connection between logic and programs where proofs are the result of computation rather than the starting point as in Part II. This gives rise to the paradigm of *logic programming* where the process of computation is one of systematic proof search. Depending on how we search for proofs, different kinds of algorithms can be described at a very high level of abstraction. Exercises in this part focus on exploiting logic programming to implement various algorithms in concrete languages such as Prolog.

In Part IV we study logics with more general and more refined notions of truth. For example, in temporal logic we are concerned with reasoning about truth relative to time. Another example is the modal logic  $S_5$  where we reason about truth in a collection of worlds, each of which is connected to all other worlds. Proofs in this logic can be given an interpretation as dis-

tributed computation. Similarly, *linear logic* is a substructural logic where truth is ephemeral and may change in the process of deduction. As we will see, this naturally corresponds to imperative programming.

### 3 Goals

There are several related goals for this course. The first is simply that we would like students to gain a good working knowledge of constructive logic and its relation to computation. This includes the translation of informally specified problems to logical language, the ability to recognize correct proofs and construct them.

The second set of goals concerns the transfer of this knowledge to other kinds of reasoning. We will try to illuminate logic and the underlying philosophical and mathematical principles from various points of view. This is important, since there are many different kinds of logics for reasoning in different domains or about different phenomena<sup>1</sup>, but there are relatively few underlying philosophical and mathematical principles. Our second goal is to teach these principles so that students can apply them in different domains where rigorous reasoning is required.

A third set of goals relates to specific, important applications of logic in the practice of computer science. Examples are the design of type systems for programming languages, specification languages, or verification tools for various classes of systems. While we do not aim at teaching the use of particular systems or languages, students should have the basic knowledge to quickly learn them, based on the materials presented in this class.

These learning goals present different challenges for students from different disciplines. Lectures, recitations, exercises, and the study of these notes are all necessary components for reaching them. These notes do not cover all aspects of the material discussed in lecture, but provide a point of reference for definitions, theorems, and motivating examples. Recitations are intended to answer students' questions and practice problem solving skills that are critical for the homework assignments. Exercises are a combination of written homework to be handed in at lecture and theorem proving or programming problems to be submitted electronically using the software written in support of the course. A brief tutorial and manual are available with the on-line course material.

---

<sup>1</sup>for example: classical, intuitionistic, modal, second-order, temporal, belief, linear, relevance, affirmation, . . .

## 4 Intuitionism

We call a logic *constructive* if its proofs describe effective constructions. The emphasis here is on *effective* which is to say that the construction conveyed by a proof can actually be carried out mechanically. In other words, constructive proofs describe algorithms. At first one might think that all proofs describe constructions of this form, and this was historically true for a long time. At some point in the 19th century this direct link between mathematics and computation seemed to get lost. Some mathematicians objected to this and started to develop a foundations of mathematics in which all proofs denote effective constructions.

In order to understand this distinction better, we start with a theorem that illustrates the distinction, the so-called *Banach-Tarski Paradox*.<sup>2</sup>

**Theorem 1** *Given a solid ball in 3-dimensional space, there exists a decomposition of the ball into a finite number of disjoint subsets, which can then be put back together in a different way to yield two identical copies of the original ball. Indeed, the reassembly process involves only moving the pieces around and rotating them, without changing their shape. The reconstruction can work with as few as five pieces.*

This is considered paradoxical, since we obviously cannot carry out such a decomposition. The intermediate pieces are in fact non-measurable infinite scatterings of points. The decomposition relies critically on the axiom of choice in set theory, which is highly non-constructive.

This is the kind of theorem (and proof, which we not show here but is sketched in the article) that mathematician L.E.J. Brouwer<sup>3</sup> might have objected to. It is meaningless with respect to our understanding of effective constructions, even if the formalities of its proof are sound. This entails a criticism of Hilbert's program, who posited that at the foundations of mathematics should be a formal system of axioms and inference rules with respect to which we can judge the correctness of mathematical arguments. Brouwer called himself an *intuitionist*, perhaps to contrast himself to Hilbert as a *formalist*.<sup>4</sup> Since intuitionistic logic has subsequently also been formalized (e.g., by Kolmogorov and Heyting), the modern way of framing the opposing sides are *intuitionistic logic* (or arithmetic) and *classical logic* (or arithmetic).

---

<sup>2</sup>See [https://en.wikipedia.org/wiki/Banach-Tarski\\_paradox](https://en.wikipedia.org/wiki/Banach-Tarski_paradox)

<sup>3</sup>See [https://en.wikipedia.org/wiki/L.\\_E.\\_J.\\_Brouwer](https://en.wikipedia.org/wiki/L._E._J._Brouwer)

<sup>4</sup>For more on this controversy in the foundations of mathematics, see [https://en.wikipedia.org/wiki/Brouwer-Hilbert\\_controversy](https://en.wikipedia.org/wiki/Brouwer-Hilbert_controversy).

One of the key differences is the interpretation of the existential quantifier. In intuitionistic logic, proving  $\exists x. A(x)$  entails exhibiting a witness  $t$  and a proof of  $A(t)$ . In classical logic, it is sufficient to show that  $\forall x. \neg A(x)$  is impossible without exhibiting a witness in the proof.

As example, we consider the following theorem and proof.

**Theorem 2** *There are two irrational numbers  $a$  and  $b$  such that  $a^b$  is rational.*

**Proof:** Consider  $\sqrt{2}^{\sqrt{2}}$ . There are two cases:

**Case:**  $\sqrt{2}^{\sqrt{2}}$  is rational. Then  $a = b = \sqrt{2}$  satisfies the claim.

**Case:**  $\sqrt{2}^{\sqrt{2}}$  is irrational. Then  $a = \sqrt{2}^{\sqrt{2}}$  and  $b = \sqrt{2}$  satisfy the claim, since  $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$ .

□

At this point, the classical mathematician is profoundly happy, since this is an extremely short and elegant proof of a prima facie nontrivial theorem. The intuitionist is profoundly unhappy, since it does not actually exhibit irrational witnesses  $a$  and  $b$  such that  $a^b$  is rational. They might  $a = b = \sqrt{2}$ , or they might be  $a = \sqrt{2}^{\sqrt{2}}$  and  $b = \sqrt{2}$ . Therefore an intuitionist should reject this proof.

The step which turns out to be incorrect here is to assume that there are two cases (either  $\sqrt{2}^{\sqrt{2}}$  is rational or not) without knowing which of the cases hold. More generally, an intuitionist rejects the *law of excluded middle* that any proposition is either true or false (in symbols:  $A \vee \neg A$ ). Concretely, what counts as a constructive proof of  $A \vee B$  is either a proof of  $A$  or a proof of  $B$ . So in addition to existential quantification, the intuitionistic and classical mathematician disagree on the interpretation of disjunction.

However, all is not lost! As an intuitionist, I look at the above proof and say

*Oh, I understand your proof, but it is for a different theorem! What you have proven is:*

**Theorem.** *If  $\sqrt{2}^{\sqrt{2}}$  is rational or not, then there are two irrational numbers  $a$  and  $b$  such that  $a^b$  is rational.*

Surprisingly, as long as we stick to pure logic, or perhaps the theory of natural numbers, any classical proof can be reinterpreted as an intuitionistic proof of a different theorem!<sup>5</sup> This suggests that, once we accept that

---

<sup>5</sup>We may show this interpretation in a future lecture.

intuitionism can in fact also be formalized, intuitionistic and classical and no longer in conflict. Instead, intuitionistic logic is a *generalization* of classical logic in the sense that has a constructive existential quantifier and a constructive disjunction, which is absent from classical logic. At the same time, all classical theorems and proofs can be uniformly imported into intuitionistic logic under some translation.

The intuitionistic interpretation of the proof above yields another question: what is the nature of implication? The intuitionistic interpretation of this particular example clarifies this: the proof of  $A \supset B$  consists of a *function* to convert a proof of  $A$  into a proof of  $B$ . Here, this function proceeds by analyzing the proof of whether  $\sqrt{2}^{\sqrt{2}}$  is rational or not. If it is rational, we return the witnesses  $a = b = \sqrt{2}$ , together with the proof that  $a^b$  is rational in this case (which we were in fact given). If it is irrational, we return the witnesses  $a = \sqrt{2}^{\sqrt{2}}$  and  $b = \sqrt{2}$ , together with a (simple equational) proof that  $a^b = 2$  in this case.

Through this example, we have already identified three critical intuitionistic principles:

1. An intuitionistic proof of  $\exists x. A(x)$  exhibits a witness  $t$  and a proof of  $A(t)$ .
2. An intuitionistic proof of  $A \vee B$  consists of either a proof of  $A$  or a proof of  $B$ .
3. An intuitionistic proof of  $A \supset B$  contains a construction that transforms a proof of  $A$  into a proof of  $B$ .

To achieve these, an intuitionist has to reject some classical reasoning principles or axioms. In natural deduction (as discussed in Lecture 2), this is manifest in the single *axiom of excluded middle*.

As a final example, consider the claim:

**Theorem 3** *Among all the students in the class, there is a leader in the following sense: if he or she has a tattoo, then everyone in the class has a tattoo.*

In logical language, we could formalize this claim as

$$\exists x. (\text{has}(x, \text{tattoo}) \supset \forall y. \text{has}(y, \text{tattoo}))$$

where the quantifiers range of the students in this class. Here is the (non-constructive!) proof

**Proof:** Either everyone in the class has a tattoo, or there is at least one student  $s$  who does not have a tattoo.

**Case:** Everyone has a tattoo. Then any  $x$  will do<sup>6</sup>, because the conclusion of the implication holds.

**Case:** There is some student  $s$  who does not have a tattoo. Then this student  $s$  is a leader: since  $\text{has}(s, \text{tattoo})$  is false, the implication  $\text{has}(s, \text{tattoo}) \supset \forall y. \text{has}(y, \text{tattoo})$  is true.

□

This is non-constructive, because we use a form of the excluded middle to avoid naming a witness to the existential (which we cannot do without violating students' privacy in unacceptable ways). Actually, as long as the domain of quantification is non-empty (usually assumed in classical logic), this proof has nothing to do with students and tattoos, but the proof above applies to the logical form

$$\exists x. (A(x) \supset \forall y. A(y))$$

Intuitionistically, we cannot prove this without further assumptions about  $A$ .<sup>7</sup>

In the next lecture we will start to look closely at the intuitionistic meaning of the logical connectives and their proof rules, based on the interpretation we sketched in this lecture.

---

<sup>6</sup>As pointed out by a student, this requires there to be at least one person in the class, which must be the case or that student couldn't have pointed it out.

<sup>7</sup>Exercise: which particular intuitionistically true proposition does the proof above establish?

# Lecture Notes on Natural Deduction

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 2  
August 31, 2017

## 1 Introduction

The goal of this chapter is to develop the two principal notions of logic, namely *propositions* and *proofs*. There is no universal agreement about the proper foundations for these notions. One approach, which has been particularly successful for applications in computer science, is to understand the meaning of a proposition by understanding its proofs. In the words of Martin-Löf [ML96, Page 27]:

*The meaning of a proposition is determined by [...] what counts as a verification of it.*

A *verification* may be understood as a certain kind of proof that only examines the constituents of a proposition. This is analyzed in greater detail by Dummett [Dum91] although with less direct connection to computer science. The system of inference rules that arises from this point of view is *natural deduction*, first proposed by Gentzen [Gen35] and studied in depth by Prawitz [Pra65].

In this chapter we apply Martin-Löf's approach, which follows a rich philosophical tradition, to explain the basic propositional connectives. We will see later that universal and existential quantifiers and types such as natural numbers, lists, or trees naturally fit into the same framework.

---

\*Edits by André Platzer

We will define the meaning of the usual connectives of propositional logic (conjunction, implication, disjunction) by rules that allow us to infer when they should be true, so-called *introduction rules*. From these, we derive rules for the use of propositions, so-called *elimination rules*. The resulting system of *natural deduction* is the foundation of intuitionistic logic which has direct connections to functional programming and logic programming.

## 2 Judgments and Propositions

The cornerstone of Martin-Löf's foundation of logic is a clear separation of the notions of judgment and proposition. A *judgment* is something we may know, that is, an object of knowledge. A judgment is *evident* if we in fact know it.

We make a judgment such as “*it is raining*”, because we have evidence for it. In everyday life, such evidence is often immediate: we may look out the window and see that it is raining. In logic, we are concerned with situation where the evidence is indirect: we deduce the judgment by making correct inferences from other evident judgments. In other words: a judgment is evident if we have a proof for it.

The most important judgment form in logic is “*A is true*”, where *A* is a proposition. There are many others that have been studied extensively. For example, “*A is false*”, “*A is true at time t*” (from temporal logic), “*A is necessarily true*” (from modal logic), “*program M has type τ*” (from programming languages), etc.

Returning to the first judgment, let us try to explain the meaning of conjunction. We write *A true* for the judgment “*A is true*” (presupposing that *A* is a proposition). Given propositions *A* and *B*, we can form the compound proposition “*A and B*”, written more formally as  $A \wedge B$ . But we have not yet specified what conjunction *means*, that is, what counts as a verification of  $A \wedge B$ . This is accomplished by the following inference rule:

$$\frac{A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \wedge I$$

Here the name  $\wedge I$  stands for “conjunction introduction”, since the conjunction is introduced in the conclusion.

This rule allows us to conclude that  $A \wedge B$  *true* if we already know that *A true* and *B true*. In this inference rule, *A* and *B* are *schematic variables*, and  $\wedge I$  is the name of the rule. Intuitively, the  $\wedge I$  rule says that a proof of  $A \wedge B$  *true* consists of a proof of *A true* together with a proof of *B true*.

The general form of an inference rule is

$$\frac{J_1 \dots J_n}{J} \text{ name}$$

where the judgments  $J_1, \dots, J_n$  are called the *premises*, the judgment  $J$  is called the *conclusion*. In general, we will use letters  $J$  to stand for judgments, while  $A, B$ , and  $C$  are reserved for propositions.

We take conjunction introduction as specifying the meaning of  $A \wedge B$  completely. So what can be deduced if we know that  $A \wedge B$  is true? By the above rule, to have a verification for  $A \wedge B$  means to have verifications for  $A$  and  $B$ . Hence the following two rules are justified:

$$\frac{A \wedge B \text{ true}}{A \text{ true}} \wedge E_1 \quad \frac{A \wedge B \text{ true}}{B \text{ true}} \wedge E_2$$

The name  $\wedge E_1$  stands for “first/left conjunction elimination”, since the conjunction in the premise has been eliminated in the conclusion. Similarly  $\wedge E_2$  stands for “second/right conjunction elimination”. Intuitively, the  $\wedge E_1$  rule says that  $A$  *true* follows if we have a proof of  $A \wedge B$  *true*, because “we must have had a proof of  $A$  *true* to justify  $A \wedge B$  *true*”.

We will later see what precisely is required in order to guarantee that the formation, introduction, and elimination rules for a connective fit together correctly. For now, we will informally argue the correctness of the elimination rules, as we did for the conjunction elimination rules.

As a second example we consider the proposition “*truth*” written as  $\top$ . Truth should always be true, which means its introduction rule has no premises.

$$\frac{}{\top \text{ true}} \top I$$

Consequently, we have no information if we know  $\top$  *true*, so there is no elimination rule.

A conjunction of two propositions is characterized by one introduction rule with two premises, and two corresponding elimination rules. We may think of truth as a conjunction of zero propositions. By analogy it should then have one introduction rule with zero premises, and zero corresponding elimination rules. This is precisely what we wrote out above.

### 3 Hypothetical Judgments

Consider the following derivation, for arbitrary propositions  $A$ ,  $B$ , and  $C$ :

$$\frac{\frac{A \wedge (B \wedge C) \text{ true}}{B \wedge C \text{ true}} \wedge E_2}{B \text{ true}} \wedge E_1$$

Have we actually proved anything here? At first glance it seems that cannot be the case:  $B$  is an arbitrary proposition; clearly we should not be able to prove that it is true. Upon closer inspection we see that all inferences are correct, but the first judgment  $A \wedge (B \wedge C)$  true has not been justified. We can extract the following knowledge:

*From the assumption that  $A \wedge (B \wedge C)$  is true, we deduce that  $B$  must be true.*

This is an example of a *hypothetical judgment*, and the figure above is an *hypothetical deduction*. In general, we may have more than one assumption, so a hypothetical deduction has the form

$$\begin{array}{c} J_1 \quad \dots \quad J_n \\ \vdots \\ J \end{array}$$

where the judgments  $J_1, \dots, J_n$  are unproven assumptions, and the judgment  $J$  is the conclusion. All instances of the inference rules are hypothetical judgments as well (albeit possibly with 0 assumptions if the inference rule has no premises).

Many mistakes in reasoning arise because dependencies on some hidden assumptions are ignored. When we need to be explicit, we will write  $J_1, \dots, J_n \vdash J$  for the hypothetical judgment which is established by the hypothetical deduction above. We may refer to  $J_1, \dots, J_n$  as the antecedents and  $J$  as the succedent of the hypothetical judgment. For example, the hypothetical judgment  $A \wedge (B \wedge C)$  true  $\vdash B$  true is proved by the above hypothetical deduction that  $B$  true indeed follows from the hypothesis  $A \wedge (B \wedge C)$  true using inference rules.

**Substitution Principle for Hypotheses:** We can always substitute a proof for any hypothesis  $J_i$  to eliminate the assumption. Into the above hypothetical deduction, a proof of its hypothesis  $J_i$

$$\begin{array}{c} K_1 \quad \dots \quad K_m \\ \vdots \\ J_i \end{array}$$

can be substituted in for  $J_i$  to obtain the hypothetical deduction

$$\begin{array}{ccccccc} K_1 & \dots & K_m \\ \vdots & & \vdots \\ J_1 & \dots & J_i & \dots & J_n \\ \vdots & & \vdots & & \vdots \\ & & & & J \end{array}$$

This hypothetical deduction concludes  $J$  from the unproven assumptions  $J_1, \dots, J_{i-1}, K_1, \dots, K_m, J_{i+1}, \dots, J_n$  and justifies the hypothetical judgment

$$J_1, \dots, J_{i-1}, K_1, \dots, K_m, J_{i+1}, \dots, J_n \vdash J$$

That is, into the hypothetical judgment  $J_1, \dots, J_n \vdash J$ , we can always substitute a derivation of the judgment  $J_i$  that was used as a hypothesis to obtain a derivation which no longer depends on the assumption  $J_i$ . A hypothetical deduction with 0 assumptions is a *proof* of its conclusion  $J$ .

One has to keep in mind that hypotheses may be used more than once, or not at all. For example, for arbitrary propositions  $A$  and  $B$ ,

$$\frac{\frac{A \wedge B \text{ true}}{B \text{ true}} \wedge E_2 \quad \frac{A \wedge B \text{ true}}{A \text{ true}} \wedge E_1}{B \wedge A \text{ true}} \wedge I$$

can be seen a hypothetical derivation of  $A \wedge B \text{ true} \vdash B \wedge A \text{ true}$ . Similarly, a minor variation of the first proof in this section is a hypothetical derivation for the hypothetical judgment  $A \wedge (B \wedge C) \text{ true} \vdash B \wedge A \text{ true}$  that uses the hypothesis twice.

With hypothetical judgments, we can now explain the meaning of implication “ $A$  implies  $B$ ” or “if  $A$  then  $B$ ” (more formally:  $A \supset B$ ). The introduction rule reads:  $A \supset B$  is true, if  $B$  is true under the assumption that  $A$  is true.

$$\frac{\overline{A \text{ true}}^u}{\overline{A \supset B \text{ true}}} \supset I^u$$

The tricky part of this rule is the label  $u$  and its bar. If we omit this annotation, the rule would read

$$\frac{\begin{array}{c} A \text{ true} \\ \vdots \\ B \text{ true} \end{array}}{A \supset B \text{ true}} \supset I$$

which would be incorrect: it looks like a derivation of  $A \supset B$  *true* from the hypothesis  $A$  *true*. But the assumption  $A$  *true* is introduced in the process of proving  $A \supset B$  *true*; the conclusion should not depend on it! Certainly, whether the implication  $A \supset B$  is true is independent of the question whether  $A$  itself is actually true. Therefore we label uses of the assumption with a new name  $u$ , and the corresponding inference which introduced this assumption into the derivation with the same label  $u$ .

The rule makes intuitive sense, a proof justifying  $A \supset B$  *true* assumes, hypothetically, the left-hand side of the implication so that  $A$  *true*, and uses this to show the right-hand side of the implication by proving  $B$  *true*. The proof of  $A \supset B$  *true* constructs a proof of  $B$  *true* from the additional assumption that  $A$  *true*.

As a concrete example, consider the following proof of  $A \supset (B \supset (A \wedge B))$ .

$$\frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^w}{\frac{\overline{A \wedge B \text{ true}}^{\wedge I}}{\frac{B \supset (A \wedge B) \text{ true}}{A \supset (B \supset (A \wedge B)) \text{ true}}^{\supset I^u}}^{\supset I^w}}$$

Note that this derivation is not hypothetical (it does not depend on any assumptions). The assumption  $A$  *true* labeled  $u$  is discharged in the last inference, and the assumption  $B$  *true* labeled  $w$  is discharged in the second-to-last inference. It is critical that a discharged hypothesis is no longer available for reasoning, and that all labels introduced in a derivation are distinct.

Finally, we consider what the elimination rule for implication should say. By the only introduction rule, having a proof of  $A \supset B$  *true* means that we have a hypothetical proof of  $B$  *true* from  $A$  *true*. By the substitution principle, if we also have a proof of  $A$  *true* then we get a proof of  $B$  *true*.

$$\frac{A \supset B \text{ true} \quad A \text{ true}}{B \text{ true}}^{\supset E}$$

This completes the rules concerning implication.

With the rules so far, we can write out proofs of simple properties concerning conjunction and implication. The first expresses that conjunction is commutative—intuitively, an obvious property.

$$\frac{\frac{\frac{A \wedge B \text{ true}}{B \text{ true}} \wedge E_2 \quad \frac{\frac{A \wedge B \text{ true}}{A \text{ true}} \wedge E_1}{A \wedge A \text{ true}} \wedge I}{B \wedge A \text{ true}} \supset I^u}{(A \wedge B) \supset (B \wedge A) \text{ true}}$$

When we construct such a derivation, we generally proceed by a combination of bottom-up and top-down reasoning. The next example is a distributivity law, allowing us to move implications over conjunctions. This time, we show the partial proofs in each step. Of course, other sequences of steps in proof constructions are also possible.

$$\vdots \\
 (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \text{ true}$$

First, we use the implication introduction rule bottom-up.

$$\frac{\frac{\frac{A \supset (B \wedge C) \text{ true}}{\vdots} u}{(A \supset B) \wedge (A \supset C) \text{ true}} \supset I^u}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \text{ true}}$$

Next, we use the conjunction introduction rule bottom-up, copying the available assumptions to both branches in the scope.

$$\frac{\frac{\frac{A \supset (B \wedge C) \text{ true}}{\vdots} u \quad \frac{A \supset (B \wedge C) \text{ true}}{\vdots} u}{\frac{\frac{A \supset B \text{ true}}{(A \supset B) \wedge (A \supset C) \text{ true}} \quad \frac{A \supset C \text{ true}}{(A \supset B) \wedge (A \supset C) \text{ true}} \wedge I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \text{ true}} \supset I^u}}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \text{ true}}$$

We now pursue the left branch, again using implication introduction bottom-up.

$$\frac{\overline{A \supset (B \wedge C) \text{ true}}^u \quad \overline{A \text{ true}}^w}{\overline{A \supset (B \wedge C) \text{ true}}^u}$$

$$\vdots \qquad \qquad \vdots$$

$$\frac{\overline{B \text{ true}} \quad \overline{A \supset B \text{ true}}^{D\!I^w}}{\overline{A \supset B \text{ true}}^{D\!I^w}}$$

$$\frac{\overline{A \supset C \text{ true}} \quad \overline{(A \supset B) \wedge (A \supset C) \text{ true}}^{\wedge I}}{\overline{A \supset C \text{ true}}^{\wedge I}}$$

$$\frac{\overline{(A \supset B) \wedge (A \supset C) \text{ true}}^{\wedge I}}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \text{ true}^{D\!I^u}}$$

Note that the hypothesis  $A \text{ true}$  is available only in the left branch and not in the right one: it is discharged at the inference  $\supset I^w$ . We now switch to top-down reasoning, taking advantage of implication elimination.

$$\frac{\overline{A \supset (B \wedge C) \text{ true}}^u \quad \overline{A \text{ true}}^w}{\overline{B \wedge C \text{ true}}^{D\!E}}$$

$$\vdots \qquad \qquad \vdots$$

$$\frac{\overline{B \text{ true}} \quad \overline{A \supset B \text{ true}}^{D\!I^w}}{\overline{A \supset B \text{ true}}^{D\!I^w}}$$

$$\frac{\overline{A \supset C \text{ true}} \quad \overline{(A \supset B) \wedge (A \supset C) \text{ true}}^{\wedge I}}{\overline{(A \supset B) \wedge (A \supset C) \text{ true}}^{\wedge I}}$$

$$\frac{\overline{(A \supset B) \wedge (A \supset C) \text{ true}}^{\wedge I}}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \text{ true}^{D\!I^u}}$$

Now we can close the gap in the left-hand side by conjunction elimination.

$$\frac{\overline{A \supset (B \wedge C) \text{ true}}^u \quad \overline{A \text{ true}}^w}{\overline{B \wedge C \text{ true}}^{D\!E}}$$

$$\frac{\overline{B \text{ true}} \quad \overline{A \supset B \text{ true}}^{D\!I^w}}{\overline{A \supset B \text{ true}}^{D\!I^w}}$$

$$\frac{\overline{A \supset C \text{ true}} \quad \overline{(A \supset B) \wedge (A \supset C) \text{ true}}^{\wedge I}}{\overline{(A \supset B) \wedge (A \supset C) \text{ true}}^{\wedge I}}$$

$$\frac{\overline{(A \supset B) \wedge (A \supset C) \text{ true}}^{\wedge I}}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \text{ true}^{D\!I^u}}$$

The right premise of the conjunction introduction can be filled in analogously. We skip the intermediate steps and only show the final derivation.

$$\begin{array}{c}
 \frac{}{A \supset (B \wedge C) \text{ true}}^u \quad \frac{}{A \text{ true}}^w \quad \frac{}{A \supset (B \wedge C) \text{ true}}^u \quad \frac{}{A \text{ true}}^v \\
 \supset E \qquad \supset E \\
 \frac{\frac{B \wedge C \text{ true}}{B \text{ true}} \wedge E_1}{A \supset B \text{ true}} \supset I^w \qquad \frac{\frac{B \wedge C \text{ true}}{C \text{ true}} \wedge E_2}{A \supset C \text{ true}} \supset I^v \\
 \frac{}{(A \supset B) \wedge (A \supset C) \text{ true}} \wedge I \\
 \frac{}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \text{ true}} \supset I^u
 \end{array}$$

## 4 Disjunction and Falsehood

So far we have explained the meaning of conjunction, truth, and implication. The disjunction “ $A$  or  $B$ ” (written as  $A \vee B$ ) is more difficult, but does not require any new judgment forms. Disjunction is characterized by two introduction rules:  $A \vee B$  is true, if either  $A$  or  $B$  is true.

$$\frac{A \text{ true}}{A \vee B \text{ true}} \vee I_1 \qquad \frac{B \text{ true}}{A \vee B \text{ true}} \vee I_2$$

Now it would be incorrect to have an elimination rule such as

$$\frac{A \vee B \text{ true}}{A \text{ true}} \vee E_1?$$

because even if we know that  $A \vee B$  is true, we do not know whether the disjunct  $A$  or the disjunct  $B$  is true. Concretely, with such a rule we could derive the truth of *every* proposition  $A$  as follows:

$$\frac{\frac{\frac{}{\top \text{ true}} \top I}{A \vee \top \text{ true}} \vee I_2}{A \text{ true}} \vee E_1?$$

Thus we take a different approach. If we know that  $A \vee B$  is true, we must consider two cases:  $A$  true and  $B$  true. If we can prove a conclusion  $C$  true in both cases, then  $C$  must be true! Written as an inference rule:

$$\frac{\frac{\frac{A \text{ true}}{A \vee B \text{ true}}^u \quad \frac{B \text{ true}}{A \vee B \text{ true}}^w}{\vdots} \quad \frac{C \text{ true}}{C \text{ true}} \quad \frac{C \text{ true}}{C \text{ true}}}{C \text{ true}} \vee E^{u,w}$$

If we know that  $A \vee B$  true then we also know  $C$  true, if that follows both in the case where  $A \vee B$  true because  $A$  is true and in the case where  $A \vee B$  true because  $B$  is true. Note that we use once again the mechanism of hypothetical judgments. In the proof of the second premise we may use the assumption  $A$  true labeled  $u$ , in the proof of the third premise we may use the assumption  $B$  true labeled  $w$ . Both are discharged at the disjunction elimination rule.

Let us justify the conclusion of this rule more explicitly. By the first premise we know  $A \vee B$  true. The premises of the two possible introduction rules are  $A$  true and  $B$  true. In case  $A$  true we conclude  $C$  true by the substitution principle and the second premise: we substitute the proof of  $A$  true for any use of the assumption labeled  $u$  in the hypothetical derivation. The case for  $B$  true is symmetric, using the hypothetical derivation in the third premise.

Because of the complex nature of the elimination rule, reasoning with disjunction is more difficult than with implication and conjunction. As a simple example, we prove the commutativity of disjunction.

$$\vdots \\ (A \vee B) \supset (B \vee A) \text{ true}$$

We begin with an implication introduction.

$$\frac{\vdots}{\begin{array}{c} A \vee B \text{ true} \\ \vdots \\ B \vee A \text{ true} \end{array}}^u \supset I^u \\ (A \vee B) \supset (B \vee A) \text{ true}$$

At this point we cannot use either of the two disjunction introduction rules. The problem is that neither  $B$  nor  $A$  follow from our assumption  $A \vee B$ ! So first we need to distinguish the two cases via the rule of disjunction elimination.

$$\frac{\vdots}{\begin{array}{c} \overline{A \text{ true}}^v \quad \overline{B \text{ true}}^w \\ \vdots \quad \vdots \\ A \vee B \text{ true} \quad B \vee A \text{ true} \quad B \vee A \text{ true} \end{array}}_{B \vee A \text{ true}} \vee E^{v,w} \\ \frac{\vdots}{(A \vee B) \supset (B \vee A) \text{ true}} \supset I^u$$

The assumption labeled  $u$  is still available for each of the two proof obligations, but we have omitted it, since it is no longer needed.

Now each gap can be filled in directly by the two disjunction introduction rules.

$$\frac{\frac{\frac{A \text{ true}}{A \vee B \text{ true}}^u \quad \frac{\frac{A \text{ true}}{B \vee A \text{ true}}^v \vee I_2 \quad \frac{\frac{B \text{ true}}{B \vee A \text{ true}}^w \vee I_1}{B \vee A \text{ true}} \vee E^{v,w}}{B \vee A \text{ true}} \supset I^u}{(A \vee B) \supset (B \vee A) \text{ true}}$$

This concludes the discussion of disjunction. Falsehood (written as  $\perp$ , sometimes called absurdity) is a proposition that should have no proof! Therefore there are no introduction rules.

Since there cannot be a proof of  $\perp \text{ true}$ , it is sound to conclude the truth of any arbitrary proposition if we know  $\perp \text{ true}$ . This justifies the elimination rule

$$\frac{\perp \text{ true}}{C \text{ true}} \perp E$$

We can also think of falsehood as a disjunction between zero alternatives. By analogy with the binary disjunction, we therefore have zero introduction rules, and an elimination rule in which we have to consider zero cases. This is precisely the  $\perp E$  rule above.

From this it might seem that falsehood is useless: we can never prove it. This is correct, except that we might reason from contradictory hypotheses! We will see some examples when we discuss negation, since we may think of the proposition “not  $A$ ” (written  $\neg A$ ) as  $A \supset \perp$ . In other words,  $\neg A$  is true precisely if the assumption  $A \text{ true}$  is contradictory because we could derive  $\perp \text{ true}$ .

## 5 Natural Deduction

The judgments, propositions, and inference rules we have defined so far collectively form a system of *natural deduction*. It is a minor variant of a system introduced by Gentzen [Gen35] and studied in depth by Prawitz [Pra65]. One of Gentzen’s main motivations was to devise rules that model mathematical reasoning as directly as possible, although clearly in much more detail than in a typical mathematical argument.

The specific interpretation of the truth judgment underlying these rules is *intuitionistic* or *constructive*. This differs from the *classical* or *Boolean* interpretation of truth. For example, classical logic accepts the proposition  $A \vee (A \supset B)$  as true for arbitrary  $A$  and  $B$ , although in the system we have

**Introduction Rules**

$$\frac{A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \wedge I$$

$$\frac{}{\top \text{ true}} \top I$$

$$\frac{\overline{A \text{ true}}^u}{\vdots}$$

$$\frac{B \text{ true}}{A \supset B \text{ true}} \supset I^u$$

**Elimination Rules**

$$\frac{A \wedge B \text{ true}}{A \text{ true}} \wedge E_1 \quad \frac{A \wedge B \text{ true}}{B \text{ true}} \wedge E_2$$

*no  $\top E$  rule*

$$\frac{A \supset B \text{ true} \quad A \text{ true}}{B \text{ true}} \supset E$$

$$\frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^w}{\vdots}$$

$$\frac{A \text{ true}}{A \vee B \text{ true}} \vee I_1 \quad \frac{B \text{ true}}{A \vee B \text{ true}} \vee I_2 \quad \frac{A \vee B \text{ true} \quad C \text{ true} \quad C \text{ true}}{C \text{ true}} \vee E^{u,w}$$

$$\frac{\perp \text{ true}}{C \text{ true}} \perp E$$

Figure 1: Rules for intuitionistic natural deduction

presented so far this would have no proof. Classical logic is based on the principle that every proposition must be true or false. If we distinguish these cases we see that  $A \vee (A \supset B)$  should be accepted, because in case that  $A$  is true, the left disjunct holds; in case  $A$  is false, the right disjunct holds. In contrast, intuitionistic logic is based on explicit evidence, and evidence for a disjunction requires evidence for one of the disjuncts. We will return to classical logic and its relationship to intuitionistic logic later; for now our reasoning remains intuitionistic since, as we will see, it has a direct connection to functional computation, which classical logic lacks.

We summarize the rules of inference for the truth judgment introduced so far in Figure 1.

## 6 Notational Definition

So far, we have defined the meaning of the logical connectives by their introduction rules, which is the so-called *verificationist* approach. Another common way to define a logical connective is by a *notational definition*. A notational definition gives the meaning of the general form of a proposition in terms of another proposition whose meaning has already been defined. For example, we can define *logical equivalence*, written  $A \equiv B$  as  $(A \supset B) \wedge (B \supset A)$ . This definition is justified, because we already understand implication and conjunction.

As mentioned above, another common notational definition in intuitionistic logic is  $\neg A = (A \supset \perp)$ . Several other, more direct definitions of intuitionistic negation also exist, and we will see some of them later in the course. Perhaps the most intuitive one is to say that  $\neg A$  true if  $A$  false, but this requires the new judgment of falsehood.

Notational definitions can be convenient, but they can be a bit cumbersome at times. We sometimes give a notational definition and then derive introduction and elimination rules for the connective. It should be understood that these rules, even if they may be called introduction or elimination rules, have a different status from those that define a connective. In this particular case, we get the derived rules

$$\frac{\overline{\quad}^u}{A \text{ true}} \quad ; \quad \frac{\perp \text{ true}}{\neg A \text{ true}} \neg I^u \quad \frac{\neg A \text{ true} \quad A \text{ true}}{\perp \text{ true}} \neg E$$

You should convince yourself that these are indeed derived rules under the notational definition of  $\neg A$ . They also *almost* have the form of introduction and elimination rules, except that we use  $\perp$  to define  $\neg A$ , while previously we avoided using other connectives besides the one we are defining.

## References

- [Dum91] Michael Dummett. *The Logical Basis of Metaphysics*. Harvard University Press, Cambridge, Massachusetts, 1991. The William James Lectures, 1976.

- [Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.
- [ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. Notes for three lectures given in Siena, April 1983.
- [Pra65] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Stockholm, 1965.

# Lecture Notes on Proofs as Programs

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 3  
September 5, 2017

## 1 Introduction

In this lecture we investigate a computational interpretation of constructive proofs and relate it to functional programming. On the propositional fragment of logic this is called the Curry-Howard isomorphism [How80]. From the very outset of the development of constructive logic and mathematics, a central idea has been that *proofs ought to represent constructions*. The Curry-Howard isomorphism is only a particularly poignant and beautiful realization of this idea. In a highly influential subsequent paper, Per Martin-Löf [ML80] developed it further into a more expressive calculus called *type theory*.

## 2 Propositions as Types

In order to illustrate the relationship between proofs and programs we introduce a new judgment:

$M : A$       $M$  is a proof term for proposition  $A$

We presuppose that  $A$  is a proposition when we write this judgment. We will also interpret  $M : A$  as " $M$  is a program of type  $A$ ". These dual interpretations of the same judgment is the core of the Curry-Howard isomorphism. We either think of  $M$  as a syntactic term that represents the proof of

---

\*Edits by André Platzer

A *true*, or we think of  $A$  as the type of the program  $M$ . As we discuss each connective, we give both readings of the rules to emphasize the analogy.

We intend that if  $M : A$  then  $A$  *true*. Conversely, if  $A$  *true* then  $M : A$  for some appropriate proof term  $M$ . But we want something more: every deduction of  $M : A$  should correspond to a deduction of  $A$  *true* with an identical structure and vice versa. In other words we annotate the inference rules of natural deduction with proof terms. The property above should then be obvious. In that way, proof term  $M$  of  $M : A$  will correspond directly to the corresponding proof of  $A$  *true*.

**Conjunction.** Constructively, we think of a proof of  $A \wedge B$  *true* as a pair of proofs: one for  $A$  *true* and one for  $B$  *true*. So if  $M$  is a proof of  $A$  and  $N$  is a proof of  $B$ , then the pair  $\langle M, N \rangle$  is a proof of  $A \wedge B$ .

$$\frac{M : A \quad N : B}{\langle M, N \rangle : A \wedge B} \wedge I$$

The elimination rules correspond to the projections from a pair to its first and second elements to get the individual proofs back out from a pair  $M$ .

$$\frac{M : A \wedge B}{\mathbf{fst} M : A} \wedge E_1 \quad \frac{M : A \wedge B}{\mathbf{snd} M : B} \wedge E_2$$

Hence the conjunction  $A \wedge B$  proposition corresponds to the product type  $A \times B$ . And, indeed, product types in functional programming languages have the same property that conjunction propositions  $A \wedge B$  have. Constructing a pair  $\langle M, N \rangle$  of type  $A \times B$  requires a program  $M$  of type  $A$  and a program  $N$  of type  $B$  (as in  $\wedge I$ ). Given a pair  $M$  of type  $A \times B$ , its first component of type  $A$  can be retrieved by the projection  $\mathbf{fst} M$  (as in  $\wedge E_1$ ), its second component of type  $B$  by the projection  $\mathbf{snd} M$  (as in  $\wedge E_2$ ).

**Truth.** Constructively, we think of a proof of  $\top$  *true* as a unit element that carries no information.

$$\overline{\langle \rangle} : \top \top I$$

Hence  $\top$  corresponds to the unit type  $\mathbf{1}$  with one element. There is no elimination rule and hence no further proof term constructs for truth. Indeed, we have not put any information into  $\langle \rangle$  when constructing it via  $\top I$ , so cannot expect to get any information back out when trying to eliminate it.

**Implication.** Constructively, we think of a proof of  $A \supset B$  *true* as a function which transforms a proof of  $A$  *true* into a proof of  $B$  *true*.

In mathematics and many programming languages, we define a function  $f$  of a variable  $x$  by writing  $f(x) = \dots$  where the right-hand side “ $\dots$ ” depends on  $x$ . For example, we might write  $f(x) = x^2 + x - 1$ . In functional programming, we can instead write  $f = \lambda x. x^2 + x - 1$ , that is, we explicitly form a functional object by *λ-abstraction* of a variable ( $x$ , in the example).

In the concrete syntax of our Standard ML-like programming language,  $\lambda x. M$  is written and  $\text{fn } x \Rightarrow M$ . We will try to mostly use the concrete syntax, but we may slip up occasionally and write the  $\lambda$ -notation instead.

We now use the notation of  $\lambda$ -abstraction to annotate the rule of implication introduction with proof terms. In the official syntax, we label the abstraction with a proposition (writing  $\lambda u:A$ ) in order to specify the domain of a function unambiguously. In practice we will often omit the label to make expressions shorter—usually (but not always!) it can be determined from the context.

$$\frac{\overline{u : A} \quad u}{\begin{array}{c} \vdots \\ M : B \end{array}} \supset I^u$$

$$\text{fn } u \Rightarrow M : A \supset B$$

The hypothesis label  $u$  acts as a variable, and any use of the hypothesis labeled  $u$  in the proof of  $B$  corresponds to an occurrence of  $u$  in  $M$ . Notice how a constructive proof of  $B$  *true* from the additional assumption  $A$  *true* to establish  $A \supset B$  *true* also describes the transformation of a proof of  $A$  *true* to a proof of  $B$  *true*. But the proof term  $\text{fn } u \Rightarrow M$  explicitly represents this transformation syntactically as a function, instead of leaving this construction implicit by inspection of whatever the proof does.

As a concrete example, consider the (trivial) proof of  $A \supset A$  *true*:

$$\frac{\overline{A \text{ true}} \quad u}{A \supset A \text{ true}} \supset I^u$$

If we annotate the deduction with proof terms, we obtain

$$\frac{\overline{u : A} \quad u}{(\text{fn } u \Rightarrow u) : A \supset A} \supset I^u$$

So our proof corresponds to the identity function  $\text{id}$  at type  $A$  which simply returns its argument. It can be defined with the identity function  $\text{id}(u) = u$  or  $\text{id} = (\text{fn } u \Rightarrow u)$ .

Constructively, a proof of  $A \supset B \text{ true}$  is a function transforming a proof of  $A \text{ true}$  to a proof of  $B \text{ true}$ . Using  $A \supset B \text{ true}$  by its elimination rule  $\supset E$ , thus, corresponds to providing the proof of  $A \text{ true}$  that  $A \supset B \text{ true}$  is waiting for to obtain a proof of  $B \text{ true}$ . The rule for implication elimination corresponds to function application. Following the convention in functional programming, we write  $M N$  for the application of the function  $M$  to argument  $N$ , rather than the more verbose  $M(N)$ .

$$\frac{M : A \supset B \quad N : A}{M N : B} \supset E$$

What is the meaning of  $A \supset B$  as a type? From the discussion above it should be clear that it can be interpreted as a function type  $A \rightarrow B$ . The introduction and elimination rules for implication can also be viewed as formation rules for functional abstraction  $\text{fn } u \Rightarrow M$  and application  $M N$ . Forming a functional abstraction  $\text{fn } u \Rightarrow M$  corresponds to a function that accepts input parameter  $u$  of type  $A$  and produces  $M$  of type  $B$  (as in  $\supset I$ ). Using a function  $M : A \rightarrow B$  corresponds to applying it to a concrete input argument  $N$  of type  $A$  to obtain an output  $M N$  of type  $B$ .

Note that we obtain the usual introduction and elimination rules for implication if we erase the proof terms. This will continue to be true for all rules in the remainder of this section and is immediate evidence for the soundness of the proof term calculus, that is, if  $M : A$  then  $A \text{ true}$ .

As a second example we consider a proof of  $(A \wedge B) \supset (B \wedge A) \text{ true}$ .

$$\frac{\frac{\frac{A \wedge B \text{ true}}{A \wedge B \text{ true}}^u \wedge E_2 \quad \frac{A \wedge B \text{ true}}{A \text{ true}}^u \wedge E_1}{\frac{B \wedge A \text{ true}}{B \wedge A \text{ true}} \wedge I}}{(A \wedge B) \supset (B \wedge A) \text{ true}} \supset I^u$$

When we annotate this derivation with proof terms, we obtain the swap function which takes a pair  $\langle M, N \rangle$  and returns the reverse pair  $\langle N, M \rangle$ .

$$\frac{\frac{\frac{u : A \wedge B}{u : A \wedge B}^u \wedge E_2 \quad \frac{u : A \wedge B}{\text{fst } u : A}^u \wedge E_1}{\frac{\text{snd } u : B}{\langle \text{snd } u, \text{fst } u \rangle : B \wedge A} \wedge I}}{(f n u \Rightarrow \langle \text{snd } u, \text{fst } u \rangle) : (A \wedge B) \supset (B \wedge A)} \supset I^u$$

**Disjunction.** Constructively, we think of a proof of  $A \vee B$  *true* as either a proof of  $A$  *true* or  $B$  *true*. Disjunction therefore corresponds to a disjoint sum type  $A + B$  that either store something of type  $A$  or something of type  $B$ . The two introduction rules correspond to the left and right injection into a sum type.

$$\frac{M : A}{\mathbf{inl} M : A \vee B} \vee I_1 \quad \frac{N : B}{\mathbf{inr} N : A \vee B} \vee I_2$$

In the official syntax, we have annotated the injections **inl** and **inr** with propositions  $B$  and  $A$ , again so that a (valid) proof term has an unambiguous type. In writing actual programs we usually omit this annotation. When using a disjunction  $A \vee B$  *true* in a proof, we need to be prepared to handle  $A$  *true* as well as  $B$  *true*, because we don't know whether  $\vee I_1$  or  $\vee I_2$  was used to prove it. The elimination rule corresponds to a case construct which discriminates between a left and right injection into a sum types.

$$\frac{\begin{array}{c} \overline{u : A} & u \\ \vdots & \vdots \\ M : A \vee B & N : C & O : C \end{array}}{\mathbf{case} \ M \ \mathbf{of} \ \mathbf{inl} \ u \Rightarrow N \mid \mathbf{inr} \ w \Rightarrow O : C} \ \vee E^{u,w}$$

Recall that the hypothesis labeled  $u$  is available only in the proof of the second premise and the hypothesis labeled  $w$  only in the proof of the third premise. This means that the scope of the variable  $u$  is  $N$ , while the scope of the variable  $w$  is  $O$ .

**Falsehood.** There is no introduction rule for falsehood ( $\perp$ ). We can therefore view it as the empty type **0**. The corresponding elimination rule allows a term of  $\perp$  to stand for an expression of any type when wrapped with **abort**. However, there is no computation rule for it, which means during computation of a valid program we will never try to evaluate a term of the form **abort**  $M$ .

$$\frac{M : \perp}{\mathbf{abort} M : C} \perp E$$

As before, the annotation  $C$  which disambiguates the type of **abort**  $M$  will often be omitted.

**Interaction Laws.** This completes our assignment of proof terms to the logical inference rules. Now we can interpret the interaction laws we intro-

duced early as programming exercises. Consider the following distributivity law:

$$(L11a) \quad (A \supset (B \wedge C)) \supset (A \supset B) \wedge (A \supset C) \text{ true}$$

Interpreted constructively, this assignment can be read as:

Write a function which, when given a function from  $A$  to pairs of type  $B \wedge C$ , returns two functions: one which maps  $A$  to  $B$  and one which maps  $A$  to  $C$ .

This is satisfied by the following function:

$$\text{fn } u \Rightarrow \langle (\text{fn } w \Rightarrow \mathbf{fst}(u w)), (\text{fn } v \Rightarrow \mathbf{snd}(u v)) \rangle$$

The following deduction provides the evidence:

$$\frac{\begin{array}{c} \dfrac{u : A \supset (B \wedge C) \quad u \quad w : A}{u w : B \wedge C} \wedge E_1 \\ \dfrac{\mathbf{fst}(u w) : B}{\text{fn } w \Rightarrow \mathbf{fst}(u w) : A \supset B} \supset I^w \end{array} \quad \begin{array}{c} \dfrac{u : A \supset (B \wedge C) \quad v : A}{u v : B \wedge C} \wedge E_2 \\ \dfrac{\mathbf{snd}(u v) : C}{\text{fn } v \Rightarrow \mathbf{snd}(u v) : A \supset C} \supset I^v \end{array}}{\langle (\text{fn } w \Rightarrow \mathbf{fst}(u w)), (\text{fn } v \Rightarrow \mathbf{snd}(u v)) \rangle : (A \supset B) \wedge (A \supset C)} \wedge I \\ \text{fn } u \Rightarrow \langle (\text{fn } w \Rightarrow \mathbf{fst}(u w)), (\text{fn } v \Rightarrow \mathbf{snd}(u v)) \rangle : (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C)) \supset I^u$$

Programs in constructive propositional logic are somewhat uninteresting in that they do not manipulate basic data types such as natural numbers, integers, lists, trees, etc. We introduce such data types later in this course, following the same method we have used in the development of logic.

**Summary.** To close this section we recall the *guiding principles behind the assignment of proof terms to deductions*.

1. For every deduction of  $A$  *true* there is a proof term  $M$  and deduction of  $M : A$ .
2. For every deduction of  $M : A$  there is a deduction of  $A$  *true*
3. The correspondence between proof terms  $M$  and deductions of  $A$  *true* is a bijection.

### 3 Reduction

In the preceding section, we have introduced the assignment of proof terms to natural deductions. If proofs are programs then we need to explain how proofs are to be executed, and which results may be returned by a computation.

We explain the operational interpretation of proofs in two steps. In the first step we introduce a judgment of *reduction* written  $M \Rightarrow_R M'$  and read “ $M$  reduces to  $M'$ ”. In the second step, a computation then proceeds by a sequence of reductions  $M \Rightarrow_R M_1 \Rightarrow_R M_2 \dots$ , according to a fixed strategy, until we reach a value which is the result of the computation. In this section we cover reduction; we may return to reduction strategies in a later lecture.

As in the development of propositional logic, we discuss each of the connectives separately, taking care to make sure the explanations are independent. This means we can consider various sublanguages and we can later extend our logic or programming language without invalidating the results from this section. Furthermore, it greatly simplifies the analysis of properties of the reduction rules.

In general, we think of the proof terms corresponding to the introduction rules as the *constructors* and the proof terms corresponding to the elimination rules as the *destructors*.

**Conjunction.** The constructor forms a pair, while the destructors are the left and right projections. The reduction rules prescribe the actions of the projections.

$$\begin{array}{l} \mathbf{fst} \langle M, N \rangle \Rightarrow_R M \\ \mathbf{snd} \langle M, N \rangle \Rightarrow_R N \end{array}$$

These (computational) reduction rules directly corresponds to the proof term analogue of the logical reductions for the local soundness from the previous lecture. For example:

$$\frac{\begin{array}{c} M : A \quad N : B \\ \hline \langle M, N \rangle : A \wedge B \end{array}}{\mathbf{fst} \langle M, N \rangle : A} \wedge_I \Rightarrow_R M : A$$

**Truth.** The constructor just forms the unit element,  $\langle \rangle$ . Since there is no destructor, there is no reduction rule.

**Implication.** The constructor forms a function by  $\lambda$ -abstraction, while the destructor applies the function to an argument. In general, the application of a function to an argument is computed by *substitution*. As a simple example from mathematics, consider the following equivalent definitions

$$f(x) = x^2 + x - 1 \quad f = \lambda x. x^2 + x - 1$$

and the computation

$$f(3) = (\lambda x. x^2 + x - 1)(3) = [3/x](x^2 + x - 1) = 3^2 + 3 - 1 = 11$$

In the second step, we substitute 3 for occurrences of  $x$  in  $x^2 + x - 1$ , the *body of the  $\lambda$ -expression*. We write  $[3/x](x^2 + x - 1) = 3^2 + 3 - 1$ .

In general, the notation for the substitution of  $N$  for occurrences of  $u$  in  $M$  is  $[N/u]M$ . We therefore write the reduction rule as

$$(\text{fn } u \Rightarrow M) N \implies_R [N/u]M$$

We have to be somewhat careful so that substitution behaves correctly. In particular, no variable in  $N$  should be bound in  $M$  in order to avoid conflict. We can always achieve this by renaming bound variables—an operation which clearly does not change the meaning of a proof term. Again, this computational reduction directly relates to the logical reduction from the local soundness using the substitution notation for the right-hand side:

$$\frac{\begin{array}{c} \overline{u : A} & u \\ \vdots \\ M : B \end{array}}{\text{fn } u \Rightarrow M : A \supset B} \supset^{I^u} \frac{N : A}{\frac{( \text{fn } u \Rightarrow M ) N : B}{\implies_R [N/u]M}} \supset^E$$

**Disjunction.** The constructors inject into a sum types; the destructor distinguishes cases. We need to use substitution again.

$$\begin{aligned} \text{case inl } M \text{ of inl } u \Rightarrow N \mid \text{inr } w \Rightarrow O &\implies_R [M/u]N \\ \text{case inr } M \text{ of inl } u \Rightarrow N \mid \text{inr } w \Rightarrow O &\implies_R [M/w]O \end{aligned}$$

The analogy with the logical reduction again works, for example:

$$\frac{\begin{array}{c} \overline{u : A} & u & \overline{w : B} & w \\ \vdots & \vdots & \vdots & \vdots \\ \text{inl } M : A \vee B & N : C & O : C \end{array}}{\text{case inl } M \text{ of inl } u \Rightarrow N \mid \text{inr } w \Rightarrow O : C} \vee^{I_1} \frac{}{\vee^{E^{u,w}}} \implies_R [M/u]N$$

**Falsehood.** Since there is no constructor for the empty type there is no reduction rule for falsehood. There is no computation rule and we will not try to evaluate **abort**  $M$ .

This concludes the definition of the reduction judgment. Observe that the construction principle for the (computational) reductions is to investigate what happens when a destructor is applied to a corresponding constructor. This is in correspondence with how (logical) reductions for local soundness consider what happens when an elimination rule is used in succession on the output of an introduction rule (when reading proofs top to bottom). Next, we will prove some of properties of the reduction judgment.

**Example Computations.** As an example we consider a simple program for the composition of two functions. It takes a pair of two functions, one from  $A$  to  $B$  and one from  $B$  to  $C$  and returns their composition which maps  $A$  directly to  $C$ .

$$\text{comp} : ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$$

We transform the following implicit definition into our notation step-by-step:

$$\begin{aligned} \text{comp } \langle f, g \rangle (w) &= g(f(w)) \\ \text{comp } \langle f, g \rangle &= \text{fn } w \Rightarrow g(f(w)) \\ \text{comp } u &= \text{fn } w \Rightarrow (\text{snd } u)((\text{fst } u)(w)) \\ \text{comp} &= \text{fn } u \Rightarrow \text{fn } w \Rightarrow (\text{snd } u)((\text{fst } u) w) \end{aligned}$$

The final definition represents a correct proof term, as witnessed by the following deduction that directly follows the proof term.

$$\frac{\frac{\frac{\frac{u : (A \supset B) \wedge (B \supset C)}{u : (A \supset B) \wedge (B \supset C)} \wedge E_2}{\text{snd } u : B \supset C}}{\frac{\frac{\frac{u : (A \supset B) \wedge (B \supset C)}{u : (A \supset B) \wedge (B \supset C)} \wedge E_1}{\text{fst } u : A \supset B}}{\frac{w : A}{(\text{fst } u) w : B}} \supset E}}{\frac{(\text{fst } u) w : B}{(\text{snd } u)((\text{fst } u) w) : C}}{\supset I^w}}{\frac{\frac{(\text{snd } u)((\text{fst } u) w) : C}{\text{fn } w \Rightarrow (\text{snd } u)((\text{fst } u) w) : A \supset C}}{\frac{\text{fn } w \Rightarrow (\text{snd } u)((\text{fst } u) w) : A \supset C}{(\text{fn } u \Rightarrow \text{fn } w \Rightarrow (\text{snd } u)((\text{fst } u) w)) : ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)}} \supset I^u}}$$

This proof can be read off directly from the proof term we constructed above, since it directly describes the shape of the proof and the rule to apply. For example **snd**  $u$  indicates that  $\wedge E_2$  has been used on  $u$ . We could also have first conducted the proof of  $((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$  *true* in

the same way that the above proof works and then annotate the proof with proof terms.

We now verify that the composition of two identity functions reduces again to the identity function. First, we verify the typing of this application.

$$(\text{fn } u \Rightarrow \text{fn } w \Rightarrow (\text{snd } u) ((\text{fst } u) w)) \langle (\text{fn } x \Rightarrow x), (\text{fn } y \Rightarrow y) \rangle : A \supset A$$

Now we show a possible sequence of reduction steps. This is by no means uniquely determined.

$$\begin{aligned} & (\text{fn } u \Rightarrow \text{fn } w \Rightarrow (\text{snd } u) ((\text{fst } u) w)) \langle (\text{fn } x \Rightarrow x), (\text{fn } y \Rightarrow y) \rangle \\ \xrightarrow{R} & \text{fn } w \Rightarrow (\text{snd } \langle (\text{fn } x \Rightarrow x), (\text{fn } y \Rightarrow y) \rangle) ((\text{fst } \langle (\text{fn } x \Rightarrow x), (\text{fn } y \Rightarrow y) \rangle) w) \\ \xrightarrow{R} & \text{fn } w \Rightarrow (\text{fn } y \Rightarrow y) ((\text{fst } \langle (\text{fn } x \Rightarrow x), (\text{fn } y \Rightarrow y) \rangle) w) \\ \xrightarrow{R} & \text{fn } w \Rightarrow (\text{fn } y \Rightarrow y) ((\text{fn } x \Rightarrow x) w) \\ \xrightarrow{R} & \text{fn } w \Rightarrow (\text{fn } y \Rightarrow y) w \\ \xrightarrow{R} & \text{fn } w \Rightarrow w \end{aligned}$$

We see that we may need to apply reduction steps to subterms in order to reduce a proof term to a form in which it can no longer be reduced. We postpone a more detailed discussion of this until we discuss the operational semantics in full.

## 4 Summary of Proof Terms

### Judgments.

$$\begin{array}{ll} M : A & M \text{ is a proof term for proposition } A, \text{ see Figure 1} \\ M \xrightarrow{R} M' & M \text{ reduces to } M', \text{ see Figure 2} \end{array}$$

## References

- [How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.
- [ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

Constructors	Destructors
$\frac{M : A \quad N : B}{\langle M, N \rangle : A \wedge B} \wedge I$	$\frac{M : A \wedge B}{\mathbf{fst} M : A} \wedge E_1$
$\frac{}{\langle \rangle : \top} \top I$	$\frac{M : A \wedge B}{\mathbf{snd} M : B} \wedge E_2$
	no destructor for $\top$
$\frac{}{u : A} u$ $\vdots$ $\frac{M : B}{\mathbf{fn} u \Rightarrow M : A \supset B} \supset I^u$	$\frac{M : A \supset B \quad N : A}{MN : B} \supset E$
$\frac{M : A}{\mathbf{inl} M : A \vee B} \vee I_1$	$\frac{\begin{matrix} M : A \vee B & N : C \\ \vdots & \vdots \\ O : C \end{matrix}}{\mathbf{case} M \mathbf{of} \mathbf{inl} u \Rightarrow N \mid \mathbf{inr} w \Rightarrow O : C} \vee E^{u,w}$
$\frac{N : B}{\mathbf{inr} N : A \vee B} \vee I_2$	$\frac{M : \perp}{\mathbf{abort} M : C} \perp E$
no constructor for $\perp$	

Figure 1: Proof term assignment for natural deduction

$$\begin{array}{ll} \mathbf{fst} \langle M, N \rangle & \xrightarrow{R} M \\ \mathbf{snd} \langle M, N \rangle & \xrightarrow{R} N \\ \text{no reduction for } \langle \rangle \\ (\mathbf{fn} u \Rightarrow M) N & \xrightarrow{R} [N/u]M \\ \mathbf{case} \mathbf{inl} M \mathbf{of} \mathbf{inl} u \Rightarrow N \mid \mathbf{inr} w \Rightarrow O & \xrightarrow{R} [M/u]N \\ \mathbf{case} \mathbf{inr} M \mathbf{of} \mathbf{inl} u \Rightarrow N \mid \mathbf{inr} w \Rightarrow O & \xrightarrow{R} [M/w]O \\ \text{no reduction for } \mathbf{abort} \end{array}$$

Figure 2: Proof term reductions

# Lecture Notes on Harmony

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 4  
September 7, 2017

## 1 Introduction

In the verificationist definition of the logical connectives via their introduction rules we have briefly justified the elimination rules. In this lecture, we study the balance between introduction and elimination rules more closely.

We elaborate on the verificationist point of view that logical connectives are defined by their introduction rules. We show that for intuitionistic logic as presented so far, the elimination rules are in harmony with the introduction rules in the sense that they are neither too strong nor too weak. We demonstrate this via local reductions and expansions, respectively. In the second part of the lecture we make more precise what a verification is and state, without proof, the global counterparts of the local soundness and completeness properties used to justify the elimination rules.

## 2 Local Soundness and Local Completeness

In order to show that introduction and elimination rules are in harmony we establish two properties: *local soundness* and *local completeness*.

**Local soundness** shows that the elimination rules are not too strong: no matter how we apply elimination rules to the result of an introduction we cannot gain any new information. We demonstrate this by showing that we can find a more direct proof of the conclusion of an elimination than one

---

\*Edits by André Platzer

that first introduces and then eliminates the connective in question. This is witnessed by a *local reduction* of the given introduction and the subsequent elimination.

**Local completeness** shows that the elimination rules are not too weak: there is always a way to apply elimination rules so that we can reconstitute a proof of the original proposition from the results by applying introduction rules. This is witnessed by a *local expansion* of an arbitrary given derivation into one that introduces the primary connective.

Connectives whose introduction and elimination rules are in harmony in the sense that they are locally sound and complete are properly defined from the verificationist perspective. If not, the proposed connective should be viewed with suspicion. Another criterion we would like to apply uniformly is that both introduction and elimination rules do not refer to other propositional constants or connectives (besides the one we are trying to define), which could create a dangerous dependency of the various connectives on each other. As we present correct definitions we will occasionally also give some counterexamples to illustrate the consequences of violating the principles behind the patterns of valid inference.

In the discussion of each individual connective below we use the notation

$$\begin{array}{c} \mathcal{D} \\ A \text{ true} \end{array} \implies^R \begin{array}{c} \mathcal{D}' \\ A \text{ true} \end{array}$$

for the local reduction of a deduction  $\mathcal{D}$  to another deduction  $\mathcal{D}'$  of the same judgment  $A \text{ true}$ . In fact,  $\implies_R$  can itself be a higher level judgment relating two proofs,  $\mathcal{D}$  and  $\mathcal{D}'$ , although we will not directly exploit this point of view. Similarly,

$$\begin{array}{c} \mathcal{D} \\ A \text{ true} \end{array} \implies^E \begin{array}{c} \mathcal{D}' \\ A \text{ true} \end{array}$$

is the notation of the local expansion of  $\mathcal{D}$  to  $\mathcal{D}'$ .

**Conjunction.** We start with local soundness, i.e., locally reducing an elimination of a conjunction that was just introduced. Since there are two elimination rules and one introduction, we have two cases to consider, because there are two different elimination rules  $\wedge E_1$  and  $\wedge E_2$  that could follow the

$\wedge I$  introduction rule. In either case, we can easily reduce.

$$\frac{\begin{array}{c} \mathcal{D} \\ A \text{ true} \quad B \text{ true} \end{array}}{\frac{A \wedge B \text{ true}}{\frac{A \text{ true}}{\mathcal{D}}}} \wedge I$$

$$\frac{\mathcal{E}}{\frac{A \wedge B \text{ true}}{\frac{B \text{ true}}{\mathcal{E}}}} \wedge E_1 \implies_R \frac{\mathcal{D}}{A \text{ true}}$$

$$\frac{\begin{array}{c} \mathcal{D} \\ A \text{ true} \quad B \text{ true} \end{array}}{\frac{A \wedge B \text{ true}}{\frac{B \text{ true}}{\mathcal{E}}}} \wedge I$$

$$\frac{\mathcal{E}}{\frac{A \wedge B \text{ true}}{\frac{B \text{ true}}{\mathcal{E}}}} \wedge E_2 \implies_R \frac{\mathcal{D}}{B \text{ true}}$$

These two reductions justify that, after we just proved a conjunction  $A \wedge B$  to be true by the introduction rule  $\wedge I$  from a proof  $\mathcal{D}$  of  $A \text{ true}$  and a proof  $\mathcal{E}$  of  $B \text{ true}$ , the only thing we can get back out by the elimination rules is something that we have put into the proof of  $A \wedge B \text{ true}$ . This makes  $\wedge E_1$  and  $\wedge E_2$  locally sound, because the only thing we get out is  $A \text{ true}$  which already has the direct proof  $\mathcal{D}$  as well as  $B \text{ true}$  which has the direct proof  $\mathcal{E}$ . The above two reductions make  $\wedge E_1$  and  $\wedge E_2$  locally sound.

Local completeness establishes that we are not losing information from the elimination rules. Local completeness requires us to apply eliminations to an arbitrary proof of  $A \wedge B \text{ true}$  in such a way that we can reconstitute a proof of  $A \wedge B$  from the results.

$$\frac{\mathcal{D}}{A \wedge B \text{ true}} \implies_E \frac{\frac{\begin{array}{c} \mathcal{D} \\ A \wedge B \text{ true} \end{array}}{\frac{A \text{ true}}{\frac{\mathcal{D}}{A \wedge B \text{ true}}}} \wedge E_1 \quad \frac{\begin{array}{c} \mathcal{D} \\ A \wedge B \text{ true} \end{array}}{\frac{B \text{ true}}{\frac{\mathcal{D}}{B \text{ true}}}} \wedge E_2}{\frac{A \wedge B \text{ true}}{\mathcal{D}}} \wedge I$$

This local expansion shows that, collectively, the elimination rules  $\wedge E_1$  and  $\wedge E_2$  extract all information from the judgment  $A \wedge B \text{ true}$  that is needed to reprove  $A \wedge B \text{ true}$  with the introduction rule  $\wedge I$ . Remember that the hypothesis  $A \wedge B \text{ true}$ , once available, can be used multiple times, which is very apparent in the local expansion, because the proof  $\mathcal{D}$  of  $A \wedge B \text{ true}$  can simply be repeated on the left and on the right premise.

As an example where local completeness fails, consider the case where we “forget” the second/right elimination rule  $\wedge E_2$  for conjunction. The remaining rule is still locally sound, because it proves something that was put into the proof of  $A \wedge B \text{ true}$ , but not locally complete because we cannot extract a proof of  $B$  from the assumption  $A \wedge B$ . Now, for example, we cannot prove  $(A \wedge B) \supset (B \wedge A)$  even though this should clearly be true.

**Substitution Principle.** We need the defining property for hypothetical judgments before we can discuss implication. Intuitively, we can always substitute a deduction of  $A \text{ true}$  for any use of a hypothesis  $A \text{ true}$ . In order to avoid ambiguity, we make sure assumptions are labelled and we substitute for all uses of an assumption with a given label. Note that we can only substitute for assumptions that are not discharged in the subproof we are considering. The substitution principle then reads as follows:

If

$$\frac{}{A \text{ true}}^u \mathcal{E} \\ A \text{ true}$$

is a hypothetical proof of  $B \text{ true}$  under the undischarged hypothesis  $A \text{ true}$  labelled  $u$ , and

$$\frac{\mathcal{D}}{A \text{ true}}$$

is a proof of  $A \text{ true}$  then

$$\frac{\mathcal{D}}{A \text{ true}}^u \mathcal{E} \\ A \text{ true}$$

is our notation for substituting  $\mathcal{D}$  for all uses of the hypothesis labelled  $u$  in  $\mathcal{E}$ . This deduction, also sometime written as  $[\mathcal{D}/u]\mathcal{E}$  no longer depends on  $u$ .

**Implication.** To witness local soundness, we reduce an implication introduction followed by an elimination using the substitution operation.

$$\frac{\frac{\frac{\frac{}{A \text{ true}}^u}{\mathcal{E}}}{B \text{ true}} \supset I^u}{A \supset B \text{ true}} \quad \frac{\frac{\mathcal{D}}{A \text{ true}}}{A \supset B \text{ true}} \supset E}{B \text{ true}} \Rightarrow_R \frac{\frac{\mathcal{D}}{A \text{ true}}^u \mathcal{E}}{B \text{ true}} \supset R$$

The conditions on the substitution operation is satisfied, because  $u$  is introduced at the  $\supset I^u$  inference and therefore not discharged in  $\mathcal{E}$ .

Local completeness is witnessed by the following expansion.

$$\frac{\frac{\frac{\mathcal{D}}{A \supset B \text{ true}} \frac{\overline{A \text{ true}}}{\overline{B \text{ true}}} u}{\supset E} \supset I^u}{A \supset B \text{ true}} \implies_E \frac{\overline{B \text{ true}}}{\overline{A \supset B \text{ true}}} \supset I^u$$

Here  $u$  must be chosen fresh: it only labels the new hypothesis  $A \text{ true}$  which is used only once.

**Disjunction.** For disjunction we also employ the substitution principle because the two cases we consider in the elimination rule introduce hypotheses. Also, in order to show local soundness we have two possibilities for the introduction rule, in both situations followed by the only elimination rule.

$$\frac{\frac{\frac{\mathcal{D}}{A \text{ true}} \vee I_L \quad \frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^w}{\frac{\mathcal{E}}{C \text{ true}} \quad \frac{\mathcal{F}}{C \text{ true}}} \vee E^{u,w}}{C \text{ true}} \implies_R \frac{\frac{\mathcal{D}}{A \text{ true}}^u}{\frac{\mathcal{E}}{C \text{ true}}}}{C \text{ true}}$$
  

$$\frac{\frac{\frac{\mathcal{D}}{B \text{ true}} \vee I_R \quad \frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^w}{\frac{\mathcal{E}}{C \text{ true}} \quad \frac{\mathcal{F}}{C \text{ true}}} \vee E^{u,w}}{C \text{ true}} \implies_R \frac{\frac{\mathcal{D}}{B \text{ true}}^w}{\frac{\mathcal{F}}{C \text{ true}}}}{C \text{ true}}$$

An example of a rule that would not be locally sound is

$$\frac{A \vee B \text{ true}}{A \text{ true}} \vee E_1?$$

and, indeed, we would not be able to reduce

$$\frac{\frac{B \text{ true}}{A \vee B \text{ true}} \vee I_R}{\frac{A \text{ true}}{\vee E_1?}}$$

In fact we can now derive a contradiction from no assumption, which means the whole system is incorrect.

$$\frac{\frac{\frac{}{\top \text{ true}} \top I}{\perp \vee \top \text{ true}} \vee I_R}{\frac{\perp \text{ true}}{\vee E_1?}}$$

Local completeness of disjunction distinguishes cases on the known  $A \vee B$  true, using  $A \vee B$  true as the conclusion.

$$A \vee B \text{ true} \xrightarrow{\mathcal{D}} \frac{\frac{\mathcal{D}}{A \vee B \text{ true}} \quad \frac{\overline{A \text{ true}}^u}{A \vee B \text{ true}} \vee I_L \quad \frac{\overline{B \text{ true}}^w}{A \vee B \text{ true}} \vee I_R}{A \vee B \text{ true}} \vee E^{u,w}$$

Visually, this looks somewhat different from the local expansions for conjunction or implication. It looks like the elimination rule is applied last, rather than first. Mostly, this is due to the notation of natural deduction: the above represents the step from using the knowledge of  $A \vee B$  true and eliminating it to obtain the hypotheses  $A$  true and  $B$  true in the two cases.

**Truth.** The local constant  $\top$  has only an introduction rule, but no elimination rule. Consequently, there are no cases to check for local soundness: any introduction followed by any elimination can be reduced, because  $\top$  has no elimination rules.

However, local completeness still yields a local expansion: Any proof of  $\top$  true can be trivially converted to one by  $\top I$ .

$$\top \text{ true} \xrightarrow{\mathcal{D}} \frac{}{\top \text{ true}} \top I$$

**Falsehood.** As for truth, there is no local reduction because local soundness is trivially satisfied since we have no introduction rule.

Local completeness is slightly tricky. Literally, we have to show that there is a way to apply an elimination rule to any proof of  $\perp$  true so that we can reintroduce a proof of  $\perp$  true from the result. However, there will be zero cases to consider, so we apply no introductions. Nevertheless, the following is the right local expansion.

$$\perp \text{ true} \xrightarrow{\mathcal{D}} \frac{\perp \text{ true}}{\perp \text{ true}} \perp E$$

Reasoning about situation when falsehood is true may seem vacuous, but is common in practice because it corresponds to reaching a contradiction. In intuitionistic reasoning, this occurs when we prove  $A \supset \perp$  which is often abbreviated as  $\neg A$ . In classical reasoning it is even more frequent, due to the rule of proof by contradiction.

$$\begin{aligned}
 M : A \wedge B &\xrightarrow{E} \langle \mathbf{fst} M, \mathbf{snd} M \rangle \\
 M : A \supset B &\xrightarrow{E} \lambda u:A. M u \quad \text{for } u \text{ not free in } M \\
 M : \top &\xrightarrow{E} \langle \rangle \\
 M : A \vee B &\xrightarrow{E} \mathbf{case} M \mathbf{of} \mathbf{inl} u \Rightarrow \mathbf{inl}^B u \mid \mathbf{inr} w \Rightarrow \mathbf{inr}^A w \\
 M : \perp &\xrightarrow{E} \mathbf{abort}^\perp M
 \end{aligned}$$

Figure 1: Proof term expansions

### 3 Revisiting Proof Terms

We saw in the last lecture, that eliminations (destructors) applied to the result of introductions (constructor) give rise to computation in the form of a reduction. We invite you to go back and verify that these computational reductions are *exactly* the witnesses of the local reductions on proofs shown in this lecture! In other words, computational reductions on proof terms witness local soundness of the rules!

What about local completeness? It turns out that the local expansions are less relevant to computation. What they tell us, for example, is that if we need to return a pair from a function, we can always construct it as  $\langle M, N \rangle$  for some  $M$  and  $N$ . Another example would be that whenever we need to return a function, we can always construct it as  $\mathbf{fn} u \Rightarrow . M$  for some  $M$ .

We can derive what the local expansion must be by annotating the deductions witnessing local expansions *on proofs* from this lecture with proof terms. We leave this as an exercise to the reader. The left-hand side of each expansion has the form  $M : A$ , where  $M$  is an arbitrary term and  $A$  is a logical connective or constant applied to arbitrary propositions. On the right hand side we have to apply a destrutor to  $M$  and then reconstruct a term of the original type. The resulting rules can be found in Figure 1.

### 4 Logical Equivalence as a Connective

As another example we would now like to define a new connective, develop introduction and elimination rules, and check their local soundness and completeness (if they hold). First, the proposed introduction rule to

define the connective:

$$\frac{\begin{array}{c} \overline{A \text{ true}} \quad x \quad \overline{B \text{ true}} \quad y \\ \vdots \qquad \qquad \vdots \\ B \text{ true} \quad A \text{ true} \end{array}}{A \equiv B \text{ true}} \equiv I^{x,y}$$

This suggests the two eliminations rules below. If we omitted one of them, we would expect the eliminations *not* to be locally complete.

$$\frac{A \equiv B \text{ true} \quad A \text{ true}}{B \text{ true}} \equiv E_1 \quad \frac{A \equiv B \text{ true} \quad B \text{ true}}{A \text{ true}} \equiv E_2$$

There is one introduction and two eliminations, so we have to check two cases for local soundness. The first case:

$$\frac{\begin{array}{c} \overline{A \text{ true}} \quad x \quad \overline{B \text{ true}} \quad y \\ \mathcal{D} \qquad \qquad \mathcal{E} \\ B \text{ true} \quad A \text{ true} \end{array}}{A \equiv B \text{ true}} \equiv I^{x,y} \quad \frac{A \text{ true}}{B \text{ true}} \equiv E_1$$

We see that  $B \text{ true}$  is justified, because the proof  $\mathcal{D}$  ends in  $B \text{ true}$  and its hypothesis is proved by  $\mathcal{F}$ :

$$\frac{\begin{array}{c} \mathcal{F} \\ \overline{A \text{ true}} \quad x \\ \mathcal{D} \\ B \text{ true} \end{array}}{\Rightarrow_R B \text{ true}}$$

The other reduction is entirely symmetric.

$$\frac{\begin{array}{c} \overline{A \text{ true}} \quad x \quad \overline{B \text{ true}} \quad y \\ \mathcal{D} \qquad \qquad \mathcal{E} \\ B \text{ true} \quad A \text{ true} \end{array}}{A \equiv B \text{ true}} \equiv I^{x,y} \quad \frac{\begin{array}{c} \mathcal{F} \\ \overline{B \text{ true}} \quad y \\ \mathcal{D} \\ A \text{ true} \end{array}}{A \text{ true}} \Rightarrow_R B \text{ true} \quad \equiv E_2$$

The local expansion will exhibit the necessity of both elimination rules. You should go through this and construct it in stages—the final result of expansion may otherwise be a bit hard to understand.

$$\frac{\frac{\frac{\mathcal{D}}{A \equiv B \text{ true}} \quad \frac{x}{A \text{ true}}}{B \text{ true}} \equiv_{E_1} \frac{\frac{\mathcal{D}}{A \equiv B \text{ true}} \quad \frac{y}{B \text{ true}}}{A \text{ true}} \equiv_{E_2}}{A \equiv B \text{ true}} \equiv_{I^{x,y}}$$

At this point we know that, logically, the connective makes sense: it is both locally sound and complete.

Next, we should carry out a proof term assignment and the re-expression local reduction and expansions on proof terms. The local reduction should give us a rule of computation; the local expansion an extensional equality principle.

$$\frac{\frac{\frac{x : A \text{ true}}{\vdots} x \quad \frac{y : B \text{ true}}{\vdots} y}{N : B \text{ true} \quad M : A \text{ true}} \equiv_{I^{x,y}} \langle x \Rightarrow N, y \Rightarrow M \rangle : A \equiv B \text{ true}}{M : A \equiv B \text{ true} \quad N : A \text{ true}} \equiv_{E_1} \quad \frac{M : A \equiv B \text{ true} \quad N : B \text{ true}}{\circledcirc M N : B \text{ true}} \equiv_{E_2}$$

We can now annotate the local reductions and expansion with proof terms and read off:

$$\begin{aligned} &\circledcirc (\langle x \Rightarrow N, y \Rightarrow M \rangle P) \Rightarrow_R [P/x]N \\ &\circledcirc (\langle x \Rightarrow N, y \Rightarrow M \rangle P) \Rightarrow_R [P/y]M \\ &M : A \equiv B \Rightarrow_E \langle x \Rightarrow \circledcirc M x, y \Rightarrow \circledcirc N y \rangle \end{aligned}$$

Introducing new syntax for new connectives and programs can be tedious and difficult to use. Therefore, in practice, we probably wouldn't define logical equivalence as a new primitive, but use *notational definition* (as we did for negation):

$$A \equiv B \triangleq (A \supset B) \wedge (B \supset A)$$

whose meaning as a type is simply a pair of functions between the types  $A$  and  $B$ .

# Lecture Notes on Verifications

15-317: Constructive Logic  
Frank Pfenning

Lecture 5  
September 12, 2017

## 1 Introduction

The verificationist point of view, already introduced earlier in the course, is that the meaning of a logical connective should be determined by its introduction rule. From this meaning we derive and then check the soundness and completeness of the elimination rules. These “local” checks pertain only to a single connective at a time.

Under this point of view, what is the meaning of a *proposition*, of course constructed from multiple logical connectives? We say the meaning of a proposition is determined by its *verifications* [ML83]. In order to be consistent with the explanation of the connectives, a verification should therefore proceed by introduction rules. However, we also need to take the elimination rules into account because they inevitably appear in the proof of a proposition.

Intuitively, a verification should be a proof that only analyzes the constituents of a proposition. This restriction of the space of all possible proofs is necessary so that the definition is well-founded. For example, if we allowed *all* proofs, then in order to understand the meaning of  $A$ , we would have to understand the meaning of  $B \supset A$  and  $B$ , the whole verificationist approach is in jeopardy because  $B$  could be a proposition containing, say,  $A$ . But the meaning of  $A$  would then in turn depend on the meaning of  $A$ , creating a vicious cycle.

In this section we will make the structure of verifications more explicit. We write  $A\uparrow$  for the judgment “ $A$  has a verification”. Naturally, this should mean that  $A$  is true, and that the evidence for that has a special form. Even-

tually we will also establish the converse: if  $A$  is true then  $A$  has a verification. Verifications also play a helpful role in proof search, because  $A\uparrow$  limits limit how a proof of  $A$  can look like to a much more canonical form.

From the proof search perspective, the notion of verification is called *intercalation* [SB98].

Conjunction is easy to understand. A verification of  $A \wedge B$  should consist of a verification of  $A$  and a verification of  $B$ .

$$\frac{A\uparrow \quad B\uparrow}{A \wedge B\uparrow} \wedge I$$

We reuse here the names of the introduction rule, because this rule is strictly analogous to the introduction rule for the truth of a conjunction.

Implication, however, introduces a new hypothesis which is not explicitly justified by an introduction rule but just a new label. For example, in the proof

$$\frac{\frac{\overline{A \wedge B \text{ true}} \quad u}{A \text{ true}} \wedge E_1}{(A \wedge B) \supset A \text{ true}} \supset I^u$$

the conjunction  $A \wedge B$  is not justified by an introduction.

The informal discussion of proof search strategies earlier, namely to use introduction rules from the bottom up and elimination rules from the top down contains the answer. We introduce a second judgment,  $A\downarrow$  which means “ $A$  may be used”.  $A\downarrow$  should be the case when either  $A$  true is a hypothesis, or  $A$  is deduced from a hypothesis via elimination rules. Our local soundness arguments provide some evidence that we cannot deduce anything incorrect in this manner.

We now go through the connectives in turn, defining verifications and uses.

**Conjunction.** In summary of the discussion above, we obtain:

$$\frac{A\uparrow \quad B\uparrow}{A \wedge B\uparrow} \wedge I \quad \frac{A \wedge B\downarrow}{A\downarrow} \wedge E_1 \quad \frac{A \wedge B\downarrow}{B\downarrow} \wedge E_2$$

The first/left elimination rule can be read as: “If we can use  $A \wedge B$  we can use  $A$ ”, and similarly for the right elimination rule. The directions of the arrows of verifications and uses matches nicely with the direction in which we end up applying the proof rules. The  $\wedge I$  rule with all its verifications

is applied toward the top: A verification  $A \wedge B \uparrow$  of  $A \wedge B$  will continue to seek a verification  $A \uparrow$  of  $A$  as well as a verification  $B \uparrow$  of  $B$ . In contrast, the elimination rule  $\wedge E_1$  with all its uses is applied toward the bottom: If we have license  $A \wedge B \downarrow$  to use  $A \wedge B$ , then we also have license  $A \downarrow$  to use  $A$ .

**Implication.** The introduction rule creates a new hypothesis, which we may use in a proof. The assumption is therefore of the judgment  $A \downarrow$

$$\frac{\overline{A \downarrow} \quad u}{\begin{array}{c} \vdots \\ B \uparrow \\ A \supset B \uparrow \end{array}} \supset I^u$$

In order to use an implication  $A \supset B$  we first require a verification of  $A$ . Just requiring that  $A$  may be used would be too weak, as can be seen when trying to prove  $((A \supset A) \supset B) \supset B \uparrow$  (see Section 2) It should also be clear from the fact that we are not eliminating a connective from  $A$ .

$$\frac{A \supset B \downarrow \quad A \uparrow}{B \downarrow} \supset E$$

Verifications and uses meet in  $\supset I^u$  and  $\supset E$  due to the direction of the implication. A verification  $A \supset B \uparrow$  of  $A \supset B$  consists of a verification  $B \uparrow$  of  $B$  that has license  $A \downarrow$  to use the additional hypothesis  $A$ . A use  $A \supset B \downarrow$  of  $A \supset B$  gives license to use  $B \downarrow$  but only after launching a verification  $A \uparrow$  to verify that  $A$  actually holds.

**Disjunction.** The verifications of a disjunction immediately follow from their introduction rules.

$$\frac{A \uparrow}{A \vee B \uparrow} \vee I_L \quad \frac{B \uparrow}{A \vee B \uparrow} \vee I_R$$

A disjunction is used in a proof by cases, called here  $\vee E$ . This introduces two new hypotheses, and each of them may be used in the corresponding subproof. Whenever we set up a hypothetical judgment we are trying to find a verification of the conclusion, possibly with uses of hy-

potheses. So the conclusion of  $\vee E$  should be a verification.

$$\frac{\overline{A \downarrow}^u \quad \overline{B \downarrow}^w}{\begin{array}{c} \vdots \\ A \vee B \downarrow \end{array} \quad C \uparrow \quad C \uparrow} \frac{}{\vee E^{u,w}} C \uparrow$$

**Truth.** The only verification of truth is the trivial one.

$$\frac{}{\overline{\top \downarrow}^{\top I}}$$

A hypothesis  $\top \downarrow$  cannot be used because there is no elimination rule for  $\top$ .

**Falsehood.** There is no verification of falsehood because we have no introduction rule.

We can use falsehood, signifying a contradiction from our current hypotheses, to verify any conclusion. This is the zero-ary case of a disjunction.

$$\frac{\perp \downarrow}{C \uparrow} \perp E$$

One might argue that a license to use  $\perp$  should give us a license to use any arbitrary other  $C$ . But the  $\perp E$  rule restricts this such that  $\perp \downarrow$  is only used to show the  $C$  we are actually looking to verify, as in conclusion  $C \uparrow$  of  $\vee E$ .

**Atomic propositions.** How do we construct a verification of an atomic proposition  $P$ ? We cannot break down the structure of  $P$  because there is none, so we can only proceed if we already know  $P$  is true. This can only come from a hypothesis, so we have a rule that lets us use the knowledge of an atomic proposition to construct a verification.

$$\frac{P \downarrow}{P \uparrow} \downarrow \uparrow$$

This rule has a special status in that it represents a change in judgments but is not tied to a particular local connective. We call this a *judgmental rule* in order to distinguish it from the usual introduction and elimination rules that characterize the connectives.

**Global soundness.** Local soundness is an intrinsic property of each connective, asserting that the elimination rules for it are not too strong given the introduction rules. Global soundness is its counterpart for the whole system of inference rules. It says that if an arbitrary proposition  $A$  has a verification then we may use  $A$  without gaining any information. That is, for arbitrary propositions  $A$  and  $C$ :

$$\begin{array}{c} A \downarrow \\ \vdots \\ \text{If } A \uparrow \text{ and } C \uparrow \text{ then } C \uparrow. \end{array}$$

We would want to prove this using a substitution principle, except that the judgment  $A \uparrow$  and  $A \downarrow$  do not match. In the end, the arguments for local soundness will help us carry out this proof later in this course when we have progressed to sequent calculus.

**Global completeness.** Local completeness is also an intrinsic property of each connective. It asserts that the elimination rules are not too weak, given the introduction rule. Global completeness is its counterpart for the whole system of inference rules. It says that if we may use  $A$  then we can construct from this a verification of  $A$ . That is, for arbitrary propositions  $A$ :

$$\begin{array}{c} A \downarrow \\ \vdots \\ A \uparrow. \end{array}$$

Global completeness follows from local completeness rather directly by induction on the structure of  $A$ . Note how crucial it is to distinguish the verification judgment  $A \uparrow$  from the use judgment  $A \downarrow$  to be able to clearly state the goal of global completeness.

Because it can often shorten proofs, we implicitly used global completeness in our formulation of verifications in lecture. That is, we allowed

$$\frac{A \downarrow}{A \uparrow} \uparrow \downarrow$$

for arbitrary  $A$ .

Global soundness and completeness are properties of whole deductive systems. Their proof must be carried out in a mathematical *metalanguage* which makes them a bit different than the formal proofs that we have done

so far within natural deduction. Of course, we would like them to be correct as well, which means they should follow the same principles of valid inference that we have laid out so far.

There are two further properties we would like, relating truth, verifications, and uses. The first is that if  $A$  has a verification or  $A$  may be used, then  $A$  is true. This is rather evident since we have just specialized the introduction and elimination rules, except for the judgmental rule  $\downarrow\uparrow$ . But under the interpretation of verification and use as truth, this inference becomes redundant.

Significantly more difficult is the property that if  $A$  is true then  $A$  has a verification. Since we justified the meaning of the connectives from their verifications, a failure of this property would be devastating to the verificationist program. Fortunately it holds and can be proved by exhibiting a process of *proof normalization* that takes an arbitrary proof of  $A$  true and constructs a verification of  $A$ .

All these properties in concert show that our rules are well constructed, locally as well as globally. Experience with many other logical systems indicates that this is not an isolated phenomenon: we can employ the verificationist point of view to give coherent sets of rules not just for constructive logic, but for classical logic, temporal logic, spatial logic, modal logic, and many other logics that are of interest in computer science. Taken together, these constitute strong evidence that separating judgments from propositions and taking a verificationist point of view in the definition of the logical connectives is indeed a proper and useful foundation for logic.

Finally observe how verifications play a role in informing proof search by reducing the proof search space. The direction of the arrows indicates in which direction a judgment should be expanded during proof search. A verification  $A\uparrow$  needs to be verified upwards by applying its appropriate introduction rule. A license to use  $A\downarrow$  can be used downwards by applying its appropriate elimination rule. Verifications and uses meet in the judgmental rule  $\downarrow\uparrow$ . In fact, when you carefully examine the example deductions we have conducted so far, you will see that they all already ended up following the proof search order that verifications and uses mandate. What needed our creativity in proof search so far has no become systematic thanks to a distinction of whether  $A$  needs to be verified or whether  $A$  can be assumed to hold.

## 2 A Counterexample

In this section we illustrate how things may go wrong if we do not define the notation of verification correctly.

If the  $\supset E$  elimination rule would be modified to have second premise use  $A\downarrow$  instead of verification  $A\uparrow$ :

$$\frac{A \supset B\downarrow \quad A\downarrow}{B\downarrow} \supset E?$$

Then the verification of  $((A \supset A) \supset B) \supset B\uparrow$  would be stuck:

$$\frac{\overline{(A \supset A) \supset B\downarrow}^u \quad A \supset A\downarrow}{\frac{B\downarrow}{B\uparrow} \uparrow\downarrow} \supset E?$$

$$\frac{}{((A \supset A) \supset B) \supset B\uparrow} \supset I^u$$

because there is no rule that applies to  $A \supset A\downarrow$ . In contrast to the successful verification with the correct  $\supset E$  rule:

$$\frac{\overline{\overline{A\downarrow}^w \quad \overline{A\uparrow}^w}{\uparrow\downarrow}}{\frac{(A \supset A) \supset B\downarrow^u \quad \overline{A \supset A\uparrow}^w}{\frac{B\downarrow}{B\uparrow} \uparrow\downarrow} \supset I^w} \supset E$$

$$\frac{}{((A \supset A) \supset B) \supset B\uparrow} \supset I^u$$

## 3 Normal and Neutral Proof Terms

Any verification is a proof. This very easy to see, because we can traverse a verification and replace both  $A\uparrow$  and  $A\downarrow$  by  $A$  *true* and obtain a proof. The minimal required change is to collapse instances of the rule

$$\frac{A\downarrow}{A\uparrow} \downarrow\uparrow$$

into simply  $A$  *true*, because otherwise premise and conclusion of the rule would be identical.

These observations suggest that we should not need to devise a new notation for *proof terms*, just reuse them and distinguish those that constitute verifications. Indeed, we need two classes of terms, so that  $N : A^\uparrow$  ( $N$  is a verification of  $A$ ) and  $R : A^\downarrow$  ( $R$  is a justification for the use of  $A$ ). In the language of programs, these already happen to have names coming from a different tradition: terms  $N$  are called *normal* and terms  $R$  are called *neutral*. By annotating the inference rules for verifications and uses, we obtain the following grammatical characterization of these classes of terms.

$\begin{array}{l} \text{Neutral } R ::= x \\   \quad RN \\   \quad \text{fst } R \mid \text{snd } R \end{array}$	$\begin{array}{l} \text{Variable} \\ \text{Application} \\ \text{Projections} \end{array}$	$\begin{array}{l} \text{Hyp} \\ \supset E \\ \wedge E_{1,2} \end{array}$
$\begin{array}{l} \text{Normal } N ::= \text{fn } x \Rightarrow N \\   \quad (N_1, N_2) \\   \quad () \\   \quad \text{inl } N \mid \text{inr } M \\   \quad (\text{case } R \text{ of inl } x_1 \Rightarrow N_1 \mid \text{inl } x_2 \Rightarrow N_2) \\   \quad \text{abort } R \\   \quad R \end{array}$	$\begin{array}{l} \text{Function} \\ \text{Pair} \\ \text{Unit} \\ \text{Injections} \\ \text{Case} \\ \text{Abort} \\ \text{Normal Term} \end{array}$	$\begin{array}{l} \supset I \\ \wedge I \\ \top I \\ \vee I_{1,2} \\ \vee E \\ \perp E \\ \downarrow \uparrow \end{array}$

At first glance, the case and abort construct appear to be in the wrong place, but then we look back at the rules and see that they do indeed construct a verification of some  $C$ .

It is now easy to verify that a normal term (which includes all neutral terms) can never be reduced. This is why this class of terms is called normal which means as much as irreducible. For example, the general proof term  $\text{fst } (M_1, M_2)$  does not fit this grammar, because only  $\text{fst } R$  is allowed, and a neutral term  $R$  cannot be a pair.

If we go back to local reductions, this should not be surprising. A local reduction arises if an elimination is applied to the result of an introduction, but this means and elimination is directly below an introduction which is ruled out for verifications. The grammar above just documents this on proof terms.

## 4 Counting Normal Proofs

First, we observe that there is no introduction rule for  $\perp$  and therefore no verification of  $\perp$ . In other words, not every proposition has a verification.

If we assume global soundness (yet to be proved), then this implies the consistency of the logic.

As a second example, how many verifications are there of  $A \supset A$ , for a propositional variable  $A$ ? A minute of doodling will tell you there can be only one, namely:

$$\frac{\overline{A \downarrow} \quad u}{\frac{\overline{A \uparrow} \quad \downarrow \uparrow}{\frac{A \uparrow}{A \supset A \uparrow}}} \supset I^u$$

This also means there is exactly one normal term of type  $A \supset A$ :

$$\text{fn } u \Rightarrow u : A \supset A$$

Similarly, there are exactly two verifications of  $A \supset (A \supset A)$ , as we checked in lecture, and therefore also only two normal proof terms

$$\begin{aligned} \text{fn } u \Rightarrow \text{fn } w \Rightarrow u \\ \text{fn } u \Rightarrow \text{fn } w \Rightarrow w \end{aligned}$$

Taking things a step further, we see that the normal proofs of type  $A \supset (A \supset A) \supset A$  are bijection with the natural numbers:

$$\begin{aligned} \text{zero} &= \text{fn } z \Rightarrow \text{fn } s \Rightarrow z \\ \text{one} &= \text{fn } z \Rightarrow \text{fn } s \Rightarrow s(z) \\ \text{two} &= \text{fn } z \Rightarrow \text{fn } s \Rightarrow s(s(z)) \\ &\dots \end{aligned}$$

## References

- [ML83] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983.
- [SB98] Wilfried Sieg and John Byrnes. Normal natural deduction proofs (in classical logic). *Studia Logica*, 60(1):67–106, January 1998.

# Lecture Notes on Quantification

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 5  
September 14, 2017

## 1 Introduction

In this lecture, we introduce universal and existential quantification, making the transition from purely propositional logic to first-order intuitionistic logic. As usual, we follow the method of using introduction and elimination rules to explain the meaning of the connectives. An important aspect of the treatment of quantifiers is that it should be completely independent of the domain of quantification. We want to capture what is true of all quantifiers, rather than those applying to natural numbers or integers or rationals or lists or other type of data. We will therefore quantify over objects of an unspecified (arbitrary) type  $\tau$ . Whatever we derive, will of course also hold for specific domain (for example,  $\tau = \text{nat}$ ). The basic judgment connecting objects  $t$  to types  $\tau$  is  $t : \tau$ . We will refer to this judgment here, but not define any specific instances until later in the course when discussing data types. What emerges as a important judgmental principle is that of a parametric judgment and the associated substitution principle for objects.

## 2 Universal Quantification

First, universal quantification, written as  $\forall x:\tau. A(x)$  and pronounced “for all  $x$  of type  $\tau$ ,  $A(x)$ ”. Here  $x$  is a bound variable and can therefore be

---

\*Edits by André Platzer.

renamed so that  $\forall x:\tau. A(x)$  and  $\forall y:\tau. A(y)$  are equivalent. When we write  $A(x)$  we mean an arbitrary proposition which may depend on  $x$ .

For the introduction rule we require that  $A(a)$  be true for arbitrary  $a$ . In other words, the premise contains a *parametric judgment*, explained in more detail below.

$$\frac{\overline{a : \tau} \quad \vdots \quad A(a) \text{ true}}{\forall x:\tau. A(x) \text{ true}} \forall I^a$$

It is important that  $a$  be a *new* parameter, not used outside of its scope, which is the derivation between the new hypothesis  $a : \tau$  and the conclusion  $A(a) \text{ true}$ . In particular, it may not occur in  $\forall x:\tau. A(x)$ . The rule makes sense: A proof that  $A(x)$  holds for all  $x$  of type  $\tau$  considers any arbitrary  $a$  of type  $\tau$  and shows that  $A(a) \text{ true}$ . But it is important that  $a$  was indeed arbitrary and not constrained by anything other than its type  $\tau$ . Observe that the parameter  $a$  is of a different kind than the label for the assumption  $a$  in the implication introduction rule  $\supset I$ , because  $a$  is a parameter for objects of type  $\tau$  while  $u$  is a label of a proposition, and in fact the rules use different judgments. As a notational reminder for this difference, we not only use different names but also do not attach the parameter  $a$  to the rule bar.

If we think of this rule as the defining property of universal quantification, then a verification of  $\forall x:\tau. A(x)$  describes a construction by which an arbitrary  $t : \tau$  can be transformed into a proof of  $A(t) \text{ true}$ . The corresponding elimination rule  $\forall E$ , thus, accepts some  $t:\tau$  and concludes that  $A(t) \text{ true}$ :

$$\frac{\forall x:\tau. A(x) \text{ true} \quad t : \tau}{A(t) \text{ true}} \forall E$$

We must verify that  $t : \tau$  so that  $A(t)$  is a well-formed proposition. The elimination rule makes sense: if  $A(x)$  is true for all  $x$  of type  $\tau$ , and if  $t$  is a particular term of type  $\tau$ , then  $A(t)$  is true as well for this particular  $t$  of type  $\tau$ .

The local reduction uses the following *substitution principle for parametric judgments*:

$$\text{If } J(a) \text{ and } t : \tau \text{ then } \frac{\begin{array}{c} a : \tau \\ \mathcal{D} \\ \mathcal{E} \end{array} \quad \frac{}{\mathcal{E}} \quad \frac{}{t : \tau} \quad \frac{}{[t/a]\mathcal{D}}}{J(t)}$$

That is, if  $\mathcal{D}$  is a deduction deducing  $J(a)$  from the judgment  $a : \tau$  about parameter  $a$ , and if  $\mathcal{E}$  is a deduction that term  $t$  is of type  $\tau$ , we can substitute the term  $t$  for parameter  $a$  throughout the derivation  $\mathcal{D}$  to obtain the derivation on the right that no longer depends on parameter  $a$  and uses the deduction  $\mathcal{E}$  to show that  $t$  has the appropriate type. The right hand side is constructed by systematically substituting  $t$  for  $a$  in  $\mathcal{D}$  and the judgments occurring in it. As usual, this substitution must be *capture avoiding* to be meaningful. It is the substitution into the judgments themselves which distinguishes substitution for parameters from substitution for hypotheses. The substitution into the judgments is necessary here since the propositions in the judgments in  $\mathcal{D}$  may still mention parameter  $a$ , which needs to be substituted to become  $t$  instead.

The local reduction showing local soundness of universal quantification then exploits this substitution principle.

$$\frac{\frac{\frac{\overline{a : \tau}}{\mathcal{D}}}{A(a) \text{ true}} \forall I^a \quad \frac{\mathcal{E}}{t : \tau} \quad \frac{\overline{\mathcal{E}}}{\overline{[t/a]\mathcal{D}}} \quad \frac{\overline{[t/a]\mathcal{D}}}{A(t) \text{ true}}}{\frac{\forall x:\tau. A(x) \text{ true}}{\forall E}} \forall E \quad \Rightarrow_R \quad A(t) \text{ true}$$

The local expansion showing local completeness of universal quantification introduces a parameter which we can use to eliminate the universal quantifier.

$$\frac{\frac{\frac{\mathcal{D}}{\forall x:\tau. A(x) \text{ true}} \quad \frac{\overline{a : \tau}}{a : \tau}}{\forall E}}{A(a) \text{ true}} \quad \frac{\frac{\forall x:\tau. A(x) \text{ true}}{\forall I^a}}{\forall x:\tau. A(x) \text{ true}} \quad \forall I^a$$

As a simple example, consider the proof that universal quantifiers distribute over conjunction.

$$\frac{\frac{\frac{\frac{\frac{\overline{\forall x:\tau. (A(x) \wedge B(x)) \text{ true}}}{u} \quad \frac{\overline{a : \tau}}{a : \tau}}{\forall E}}{A(a) \wedge B(a) \text{ true}} \quad \frac{\frac{\overline{\forall x:\tau. (A(x) \wedge B(x)) \text{ true}}}{u} \quad \frac{\overline{b : \tau}}{b : \tau}}{\forall E}}{A(b) \wedge B(b) \text{ true}} \quad \frac{\frac{\overline{B(b) \text{ true}}}{\forall x:\tau. B(x) \text{ true}}}{\forall I^b}}{\frac{\frac{\overline{\forall x:\tau. B(x) \text{ true}}}{\forall I^b}}{\frac{\frac{\overline{(\forall x:\tau. A(x)) \wedge (\forall x:\tau. B(x)) \text{ true}}}{\wedge I}}{(\forall x:\tau. (A(x) \wedge B(x))) \supset (\forall x:\tau. A(x)) \wedge (\forall x:\tau. B(x)) \text{ true}}}}}{\supset I^u}}$$

Note how crucial it is that the parameter  $a$  in  $\forall I^a$  is new, otherwise, the rules would unsoundly prove that a predicate  $C$  that is reflexive (i.e.,  $C(x, x)$  holds for all  $x$ ,  $y$ , which is clearly not the case:

$$\frac{\frac{\frac{\frac{\frac{u}{\forall x:\tau. C(x, x) \text{ true}} \quad \overline{a : \tau}}{\forall E}{C(a, a) \text{ true}}}{\forall y:\tau. C(a, y) \text{ true}} \text{ } \forall I^a??}{\forall x:\tau. \forall y:\tau. C(x, y) \text{ true}} \text{ } \forall I^a}{(\forall x:\tau. C(x, x)) \supset (\forall x:\tau. \forall y:\tau. C(x, y)) \text{ true}} \text{ } \supset I^u$$

### 3 Existential Quantification

The existential quantifier is more difficult to specify, although the introduction rule seems innocuous enough. If there is a  $t$  of type  $\tau$  for which a proof of  $A(t)$  *true* succeeds, then there is a proof of  $\exists x:\tau. A(x)$  *true* witnessed by said  $t$ .

$$\frac{t : \tau \quad A(t) \text{ true}}{\exists x:\tau. A(x) \text{ true}} \exists I$$

The elimination rules creates some difficulties. We cannot write

$$\frac{\exists x:\tau. A(x) \text{ true}}{A(t) \text{ true}} \exists E?$$

because we do not know for which  $t$  is is the case that  $A(t)$  holds. It is easy to see that local soundness would fail with this rule, because we would prove  $\exists x:\tau. A(x)$  with one witness  $t$  and then eliminate the quantifier using another object  $t'$  about which we have no reason to believe it would satisfy  $A(t')$  *true*.

The best we can do is to assume that  $A(a)$  is true for some new parameter  $a$  that, because it is new, we do not know anything about. The scope of this assumption is limited to the proof of some conclusion  $C$  *true* which does not mention  $a$  (which must be new).

$$\frac{\overline{a : \tau} \quad \overline{A(a) \text{ true}} \quad u \quad \vdots}{\frac{\exists x:\tau. A(x) \text{ true} \quad C \text{ true}}{C \text{ true}}} \exists E^{a,u}$$

Here, the scope of the hypotheses  $a$  and  $u$  is the deduction on the right, indicated by the vertical dots. In particular,  $C$  may not depend on  $a$  since  $a$  would otherwise escape its scope. We use this crucially in the local reduction for local soundness to see that  $C$  is unaffected when substituting  $t$  for  $a$  in the proof.

$$\frac{\frac{\frac{\mathcal{D}}{t : \tau} \quad \frac{\mathcal{E}}{A(t) \text{ true}}}{\exists I} \quad \frac{\overline{a : \tau} \quad \frac{\overline{A(a) \text{ true}} \quad u}{\mathcal{F}}}{C \text{ true}}}{\exists E^{a,u}} \quad \Rightarrow_R \quad \frac{\frac{\mathcal{D}}{t : \tau} \quad \frac{\mathcal{E}}{A(t) \text{ true}} \quad u}{[t/a]\mathcal{F}}}{C \text{ true}}$$

The reduction requires two substitutions, one for a parameter  $a$  and one for a hypothesis  $u$ .

The local expansion showing local completeness is patterned after the disjunction, which also—somewhat surprisingly—uses the elimination rule below the introduction rule.

$$\frac{\mathcal{D}}{\exists x:\tau. A(x) \text{ true}} \quad \Rightarrow_E \quad \frac{\frac{\mathcal{D}}{\exists x:\tau. A(x) \text{ true}} \quad \frac{\overline{a : \tau} \quad \frac{\overline{A(a) \text{ true}} \quad u}{\exists x:\tau. A(x) \text{ true}}}{\exists I}}{\exists x:\tau. A(x) \text{ true}} \quad \exists E^{a,u}$$

As an example of quantifiers we show the equivalence of  $\forall x:\tau. A(x) \supset C$  and  $(\exists x:\tau. A(x)) \supset C$ , where  $C$  does not depend on  $x$ . Generally, in our propositions, any possible dependence on a bound variable is indicated by writing a general *predicate*  $A(x_1, \dots, x_n)$ . We do not make explicit when such propositions are well-formed, although appropriate rules for explicit  $A$  could be given.

When looking at a proof, the static representation on the page is an inadequate image for the dynamics of proof construction. As we did earlier, we give two examples where we show the various stages of proof construction.

$$\vdots \\ ((\exists x:\tau. A(x)) \supset C) \supset \forall x:\tau. (A(x) \supset C) \text{ true}$$

The first three steps can be taken without hesitation, because we can always apply implication and universal introduction from the bottom up without

possibly missing a proof.

$$\begin{array}{c}
 \frac{\overline{(\exists x:\tau. A(x)) \supset C \text{ true}} \quad u \quad \overline{a : \tau} \quad \overline{A(a) \text{ true}} \quad w}{\vdots} \\
 \frac{\frac{\overline{C \text{ true}}}{A(a) \supset C \text{ true}} \supset I^w}{\frac{\overline{\forall x:\tau. A(x) \supset C \text{ true}}}{((\exists x:\tau. A(x)) \supset C) \supset \forall x:\tau. (A(x) \supset C) \text{ true}}} \forall I^a \\
 \frac{\vdots}{((\exists x:\tau. A(x)) \supset C) \supset \forall x:\tau. (A(x) \supset C) \text{ true}} \supset I^u
 \end{array}$$

At this point the conclusion is atomic, so we must apply an elimination to an assumption if we follow the strategy of *introductions bottom-up* and *eliminations top-down*. The only possibility is implication elimination, since  $a : \tau$  and  $A(a) \text{ true}$  are atomic. This gives us a new subgoal.

$$\begin{array}{c}
 \frac{\overline{a : \tau} \quad \overline{A(a) \text{ true}} \quad w}{\vdots} \\
 \frac{\overline{(\exists x:\tau. A(x)) \supset C \text{ true}} \quad u \quad \overline{\exists x:\tau. A(x)}}{\frac{\overline{C \text{ true}}}{\frac{\overline{A(a) \supset C \text{ true}}}{\frac{\overline{\forall x:\tau. A(x) \supset C \text{ true}}}{((\exists x:\tau. A(x)) \supset C) \supset \forall x:\tau. (A(x) \supset C) \text{ true}}}} \supset E} \\
 \frac{\vdots}{((\exists x:\tau. A(x)) \supset C) \supset \forall x:\tau. (A(x) \supset C) \text{ true}} \supset I^u
 \end{array}$$

At this point it is easy to see how to complete the proof with an existential introduction.

$$\begin{array}{c}
 \frac{\overline{(\exists x:\tau. A(x)) \supset C \text{ true}} \quad u \quad \frac{\overline{a : \tau} \quad \overline{A(a) \text{ true}} \quad w}{\exists I}}{\frac{\overline{\exists x:\tau. A(x)}}{\frac{\overline{C \text{ true}}}{\frac{\overline{A(a) \supset C \text{ true}}}{\frac{\overline{\forall x:\tau. A(x) \supset C \text{ true}}}{((\exists x:\tau. A(x)) \supset C) \supset \forall x:\tau. (A(x) \supset C) \text{ true}}}} \supset E}} \\
 \frac{\vdots}{((\exists x:\tau. A(x)) \supset C) \supset \forall x:\tau. (A(x) \supset C) \text{ true}} \supset I^u
 \end{array}$$

We now consider the reverse implication.

$$\begin{array}{c}
 \vdots \\
 ((\forall x:\tau. (A(x) \supset C)) \supset ((\exists x:\tau. A(x)) \supset C) \text{ true})
 \end{array}$$

From the initial goal, we can blindly carry out two implication introductions, bottom-up, which yields the following situation.

$$\frac{\begin{array}{c} \overline{\exists x:\tau. A(x) \text{ true}}^w \quad \overline{\forall x:\tau. A(x) \supset C \text{ true}}^u \\ \vdots \\ \frac{C \text{ true}}{(\exists x:\tau. A(x)) \supset C \text{ true}} \supset I^w \\ \hline (\forall x:\tau. (A(x) \supset C)) \supset ((\exists x:\tau. A(x)) \supset C) \text{ true} \end{array}}{\supset I^u}$$

Now we have two choices: existential elimination applied to  $w$  or universal elimination applied to  $u$ . However, we have not introduced any terms, so only the existential elimination can go forward.

$$\frac{\begin{array}{c} \overline{\forall x:\tau. A(x) \supset C \text{ true}}^u \quad \overline{a : \tau} \quad \overline{A(a) \text{ true}}^v \\ \vdots \\ \frac{\begin{array}{c} \overline{\exists x:\tau. A(x) \text{ true}}^w \\ \hline \frac{C \text{ true}}{(\exists x:\tau. A(x)) \supset C \text{ true}} \supset I^w \end{array}}{\exists E^{a,v}} \\ \hline (\forall x:\tau. (A(x) \supset C)) \supset ((\exists x:\tau. A(x)) \supset C) \text{ true} \end{array}}{\supset I^u}$$

At this point we need to apply another elimination rule to an assumption. We don't have much to work with, so we try universal elimination.

$$\frac{\begin{array}{c} \overline{\forall x:\tau. A(x) \supset C \text{ true}}^u \quad \overline{a : \tau} \\ \hline \frac{A(a) \supset C \text{ true}}{\forall E \quad \overline{A(a) \text{ true}}^v} \end{array}}{\vdots}$$

$$\frac{\begin{array}{c} \overline{\exists x:\tau. A(x) \text{ true}}^w \\ \hline \frac{\begin{array}{c} \overline{C \text{ true}} \\ \hline \frac{(\exists x:\tau. A(x)) \supset C \text{ true}}{\exists E^{a,v}} \supset I^w \end{array}}{\hline (\forall x:\tau. (A(x) \supset C)) \supset ((\exists x:\tau. A(x)) \supset C) \text{ true} \end{array}}{\supset I^u}$$

Now we can fill the gap with an implication elimination.

$$\frac{\frac{\frac{\frac{\frac{\overline{\forall x:\tau. A(x) \supset C \text{ true}}^u}{\exists x:\tau. A(x) \text{ true}}^w \quad \overline{a : \tau}^u}{A(a) \supset C \text{ true}}^v \quad \overline{\overline{A(a) \text{ true}}^v}{\supset E}}^v}{C \text{ true}}^v \quad \overline{\exists x:\tau. A(x)) \supset C \text{ true}}^w}{\supset I^w}^w \\
 \frac{\frac{\frac{\overline{\forall x:\tau. (A(x) \supset C)) \supset ((\exists x:\tau. A(x)) \supset C) \text{ true}}^u}{(\forall x:\tau. (A(x) \supset C)) \supset ((\exists x:\tau. A(x)) \supset C) \text{ true}}^u}{\supset I^u}^u$$

Finally, note again how crucial it is that the parameter  $a$  is actually new and does not occur in the conclusion  $C$ , otherwise we could prove unsound things:

$$\frac{\frac{\frac{\overline{\exists x:\tau. C(x) \text{ true}}^u \quad \overline{a : \tau}^u \quad \overline{C(a) \text{ true}}^w}{C(a) \text{ true}}^w \quad \overline{\forall x:\tau. C(x) \supset C(a) \text{ true}}^w}{\supset I^u}^u}{\forall E^{a,w}??}$$

## 4 Verifications and Uses

In order to formalize the proof search strategy, we use the judgments  $A$  has a verification ( $A \uparrow$ ) and  $A$  may be used ( $A \downarrow$ ) as we did in the propositional case. Universal quantification is straightforward:

$$\frac{\frac{\overline{a : \tau}^u \quad \vdots}{A(a) \uparrow} \quad \forall I^a}{\forall x:\tau. A(x) \uparrow} \quad \frac{\frac{\overline{\forall x:\tau. A(x) \downarrow}^u \quad t : \tau}{A(t) \downarrow} \quad \forall E}{A(t) \downarrow}$$

We do not assign a direction to the judgment for typing objects,  $t : \tau$ .

Verifications for the existential elimination are patterned after the disjunction: we translate a usable  $\exists x:\tau. A(x)$  into a usable  $A(a)$  with a limited scope, both in the verification of some  $C$ .

$$\frac{\frac{\overline{t : \tau}^u \quad \overline{A(t) \uparrow}^u}{\exists I}^u}{\exists x:\tau. A(x) \uparrow} \quad \frac{\frac{\overline{\exists x:\tau. A(x) \downarrow}^u \quad \overline{C \uparrow}^u}{C \uparrow} \quad \exists E^{a,u}}{C \uparrow}$$

As before, the fact that every true proposition has a verification is a kind of global version of the local soundness and completeness properties. If we take this for granted (since we do not prove it until later), then we can use this to demonstrate that certain propositions are not true, parametrically.

For example, we show that  $(\exists x:\tau. A(x)) \supset (\forall x:\tau. A(x))$  is not true in general. After the first two steps of constructing a verification, we arrive at

$$\frac{\overline{\exists x:\tau. A(x) \downarrow} \quad u \quad \overline{a : \tau}}{\begin{array}{c} \vdots \\ \overline{A(a) \uparrow} \\ \overline{\forall x:\tau. A(x) \uparrow} \end{array}} \forall I^a$$

$$\frac{\overline{(\exists x:\tau. A(x)) \supset (\forall x:\tau. A(x)) \uparrow}}{\supset I^u}$$

At this point we can only apply existential elimination, which leads to

$$\frac{\overline{b : \tau} \quad \overline{A(b) \downarrow} \quad v \quad \overline{a : \tau}}{\begin{array}{c} \vdots \\ \overline{\exists x:\tau. A(x) \downarrow} \quad u \\ \overline{A(a) \uparrow} \end{array}} \exists E^{b,v}$$

$$\frac{\overline{A(a) \uparrow} \quad \overline{\forall x:\tau. A(x) \uparrow}}{\overline{(\exists x:\tau. A(x)) \supset (\forall x:\tau. A(x)) \uparrow}} \forall I^a$$

$$\frac{\overline{(\exists x:\tau. A(x)) \supset (\forall x:\tau. A(x)) \uparrow}}{\supset I^u}$$

We cannot close the gap, because  $a$  and  $b$  are different parameters. We can only apply existential elimination to assumption  $u$  again. But this only creates  $c : \tau$  and  $A(c) \downarrow$  for some new  $c$ , so have made no progress. No matter how often we apply existential elimination, since the parameter introduced must be new, we can never prove  $A(a)$ .

## 5 Proof Terms

Going back to our very first lecture, we think of an intuitionistic proof of  $\forall x:\tau. \exists y:\sigma. A(x, y)$  as exhibiting a function that, for every  $x:\tau$  constructs a witness  $y:\sigma$  and a proof that  $A(x, y)$  is true.

So the proof term for a universal quantifier should be a function and for an existential quantifier a pair consisting of a witness and a proof that the witness is correct.

We do not invent a new notation here, but reuse the notation for functions and applications.

$$\frac{\overline{a : \tau} \quad \vdots \quad M : A(a)}{(\text{fn } a \Rightarrow M) : \forall x:\tau. A(x)} \forall I^a \quad \frac{M : \forall x:\tau. A(x) \quad t : \tau}{M t : A(t)} \forall E$$

Note that the proof term  $M$  can of course depend on  $c$ , but we explicitly mark dependency only in propositions. The local reduction and expansions straightforwardly adapt the previous rules for functions.

$$(\text{fn } a \Rightarrow M) t \implies_R [t/a]M \\ M : \forall x:\tau. A(x) \implies_E (\text{fn } a \Rightarrow M a) \text{ for } a \text{ not in } M$$

You should be able to correlate these reductions with the local reductions and expansions on proofs given earlier in this lecture.

For existential introduction the proof term is a pair, but the existential elimination is an interesting case because it does not just extract the first and second component of this pair. Instead, we have a new form that names the components of the pair, following the shape of the elimination rule.

$$\frac{\overline{a : \tau} \quad \overline{u : A(a)} \quad u \quad \vdots}{(t, M) : \exists x:\tau. A(x)} \exists I \quad \frac{M : \exists x:\tau. A(x) \quad N : C}{(\text{let } (a, u) = M \text{ in } N) : C} \exists E^{a,u}$$

The local reduction will decompose the pair as expected; the reduction decomposes it and then puts it back together.

$$\text{let } (a, u) = (t, M) \text{ in } N \implies_R [M/u][t/a]N \\ M : \exists x:\tau. A(x) \implies_E \text{let } (a, u) = M \text{ in } (a, u)$$

## 6 Rule Summary

$$\begin{array}{c}
 \frac{\overline{a : \tau} \quad \vdots}{A(a) \text{ true}} \forall I^a \quad \frac{\forall x:\tau. A(x) \text{ true} \quad t : \tau}{A(t) \text{ true}} \forall E \\
 \\ 
 \frac{\overline{a : \tau} \quad \overline{A(a) \text{ true}} \quad u \quad \vdots}{t : \tau \quad A(t) \text{ true}} \exists I \quad \frac{\exists x:\tau. A(x) \text{ true} \quad C \text{ true}}{C \text{ true}} \exists E^{a,u}
 \end{array}$$

# Lecture Notes on Induction and Recursion

15-317: Constructive Logic  
Frank Pfenning

Lecture 7  
September 19, 2017

## 1 Introduction

At this point in the course we have developed a good formal understanding how *propositional intuitionistic logic* is connected to computation: propositions are types, proofs are programs, and proof reduction is computation. We have also introduced the universal and existential quantifiers, but without some data types such as natural numbers, integers, lists, trees, etc. we cannot reason about or compute with such data.

Before we formalize this, we devote today's lecture to develop some *informal* understanding of how to reason constructively with such types, and what the programs look like that correspond to such proofs.

Briefly, we reason about data using *induction*. The computational content of such proofs will be functions defined by *recursion*.

We now go through several examples, presenting proofs in the usual mathematical style and their computational contents as recursive functions.

## 2 Example: Integer Square Root

At first, we might think of specifying the square root with the theorem

$$\forall x:\text{nat}. \exists y:\text{nat}. y^2 = x$$

This is a great place to start: if we could prove that, the function extracted would take an integer  $x$  as an argument and return a witness  $y = \sqrt{x}$  and a proof that indeed  $y^2 = x$ . Unfortunately, this is not a theorem because not

every natural number is a square. So we have to allow for rounding down or up. The following specification

$$\forall x:\text{nat. } \exists y:\text{nat. } y^2 \leq x \wedge x < (y + 1)^2$$

does in fact round down. For example, for  $x = 10$  the witness  $y = 3$  will have the property  $3^2 = 9 \leq 10 \wedge 10 < 16 = (3 + 1)^2$ .

Now we have to think about how to prove this. The natural attempt is to prove it direction by *mathematical induction* on  $x$ . This means to prove it for  $x = 0$ , then assume the theorem for  $x$  and prove it for  $x + 1$ . This form of induction is also called *weak induction* because we have the induction hypothesis. We will encounter *complete induction*, also called *strong induction* later in this lecture.

**Theorem 1 (Integer Square Root)**  $\forall x:\text{nat. } \exists y:\text{nat. } y^2 \leq x \wedge x < (y + 1)^2$

**Proof:** By mathematical induction on  $x$ .

**Base:**  $x = 0$ . Then we pick the witness  $y = 0$  because  $0^2 = 0 \leq 0 \wedge 0 < 1 = (0 + 1)^2$ .

**Step:** Assume  $\exists y:\text{nat. } y^2 \leq x \wedge x < (y + 1)^2$ .

We have to prove  $\exists y:\text{nat. } y^2 \leq x + 1 \wedge x + 1 < (y + 1)^2$ .

$$\begin{aligned} \exists y:\text{nat. } y^2 \leq x \wedge x < (y + 1)^2 && \text{by i.h.} \\ a^2 \leq x \wedge x < (a + 1)^2 && \text{for some } a, \text{ by } \exists E \end{aligned}$$

Now we distinguish two cases:  $x + 1$  is still less than  $(a + 1)^2$  or it is greater than or equal to  $(a + 1)^2$ . Note that if we are given  $x$  and  $a$ , we can decide this inequality so the case distinction is constructively permissible.

**Case:**  $x + 1 < (a + 1)^2$ . Then we pick the witness  $y = a$  because

$$\begin{aligned} a^2 \leq x + 1 && \text{since } a^2 \leq x \\ x + 1 < (a + 1)^2 && \text{this case} \end{aligned}$$

**Case:**  $x + 1 \geq (a + 1)^2$ . Then we pick the witness  $y = a + 1$  because

$$\begin{aligned} x + 1 = (a + 1)^2 && \text{since } x < (a + 1)^2 \\ (a + 1)^2 \leq x + 1 && \text{from previous line} \\ x + 1 < (a + 2)^2 && \text{since } x + 1 = (a + 1)^2 < (a + 2)^2 \end{aligned}$$

□

What is the computational content of this proof? It is a recursive function, where an appeal the induction hypothesis corresponds to a recursive call. When a witness for an existential is exhibited in the proof, we return this witness. We ignore here the attendant proof that the witness is in fact correct, so the function below will have only a portion of all of the information of the proof.

```
fun isqrt 0 = 0
| isqrt (x+1) =
  let val a = isqrt x
  in if x+1 < (a+1)*(a+1)
     then a
     else a+1
  end
```

This does not literally work in Standard ML because we cannot pattern-match against  $x + 1$ , we we have to rewrite this slightly. Also, we are using the built-in type `int` instead of `nat`.

```
fun isqrt 0 = 0
| isqrt x = (* x > 0 *)
  let val a = isqrt (x-1)
  in if x < (a+1)*(a+1)
     then a
     else a+1
  end
```

This algorithm is not what one would immediately think of as an implementation of a square root. To compute the integer square root of  $x$  it runs through all the numbers up to  $x$ , essentially adding 1 every time we hit the next square (the `else` case of the conditional). Computationally this is expensive in time. It is also expensive in space because the function is not tail recursive.

We can provide a different proof that corresponds to a more efficient function (see 5).

### 3 Example: Exponentiation

We define the mathematical function of exponentiation on natural numbers by  $b^0 = 1$  and  $b^{n+1} = b \times b^n$  for  $n > 0$ . We can now prove a somewhat uninteresting theorem that there is an implementation of that.

**Theorem 2 (Exponentiation)**  $\forall b:\text{nat. } \forall n:\text{nat. } \exists y:\text{nat. } y = b^n$

**Proof:** By mathematical induction on  $n$ .

**Base:**  $n = 0$ . Then pick  $y = 1$  because  $1 = b^0$ .

**Step:** Assume  $\exists y:\text{nat. } y = b^n$ . We have to show that  $\exists y:\text{nat. } y = b^{n+1}$ .

$$\begin{array}{ll}
 \exists y:\text{nat. } y = b^n & \text{induction hypothesis} \\
 a = b^n & \text{for some } a, \text{ by } \exists E \\
 \text{Pick } y = b \times a = b \times b^n = b^{n+1} & \\
 \exists y:\text{nat. } y = b^{n+1} & \text{by } \exists I
 \end{array}$$

□

The extracted function corresponding to this proof entirely straightforward. We write it directly in Standard ML form.

```

fun exp b 0 = 1
| exp b n = (* n > 0 *)
  let val a = exp b (n-1)
  in b * a end
  
```

Again, this function is not tail-recursive since we take the result  $a$  returned by the recursive call and multiply it by  $b$ .

To obtain a tail-recursive version, we need to find a different proof of the same specification! From our experience in functional programming we know that we need to carry along an accumulator for the result in an auxiliary function. Such an auxiliary function corresponds to a *lemma* on the mathematical side. The accumulator  $c$  is an additional argument, so the lemma has one additional quantifier.

$$\forall b:\text{nat. } \forall n:\text{nat. } \forall c:\text{nat. } \exists y:\text{nat. } ???$$

The deep question is what does the lemma express? Because we *multiply* the accumulator by the base  $b$  at every recursive call, the generalization is also stated multiplicatively. In general, though, coming up with an appropriate generalization of the theorem is a creative and difficult task.

**Lemma 3**  $\forall b:\text{nat. } \forall n:\text{nat. } \forall c:\text{nat. } \exists y:\text{nat. } y = c \times b^n$

**Proof:** By mathematical induction on  $n$ .

**Base:**  $n = 0$ . Then pick  $y = c$  because  $y = c = c \times b^0$ .

**Step:** Assume  $\forall c:\text{nat. } \exists y:\text{nat. } y = c \times b^n$ . We have to show  $\forall c:\text{nat. } \exists y:\text{nat. } y = c \times b^{n+1}$ , that is,  $\exists y:\text{nat. } y = c_1 \times b^{n+1}$  for an arbitrary  $c_1$ .

$$\begin{array}{ll}
 \forall c:\text{nat. } \exists y:\text{nat. } y = c \times b^n & \text{induction hypothesis} \\
 \exists y:\text{nat. } y = (c_1 \times b) \times b^n & \text{using } c = c_1 \times b, \text{ by } \forall E \\
 a = (c_1 \times b) \times b^n = c_1 \times b^{n+1} & \text{for some } a, \text{ by } \exists E \\
 \exists y:\text{nat. } y = c_1 \times b^{n+1} & \text{picking } y = a \text{ and } \exists I \\
 \forall c:\text{nat. } \exists y:\text{nat. } y = c \times b^{n+1} & \text{by } \forall I
 \end{array}$$

□

Now the theorem no longer requires a proof by induction, directly calling on the lemma instead.

**Theorem 4**  $\forall b:\text{nat. } \forall n:\text{nat. } \exists y:\text{nat. } y = b^n$

**Proof:** From the preceding lemma by using  $c = 1$  since  $1 \times b^n = b^n$  □

The computational content is now two functions, `exp2_aux` corresponding to the lemma, and `exp2` for the theorem.

```

fun exp2_aux b 0 c = c
| exp2_aux b n c = (* n > 0 *)
  let val a = exp2_aux b (n-1) (c*b)
  in a end

fun exp2 b n = exp2_aux b n 1

```

The auxiliary function is now tail recursive because the witness  $a$  for  $y$  in the proof is just the witness from the appeal to the induction hypothesis. We can shorten the program slightly to make this more immediate:

```

fun exp2_aux b 0 c = c
| exp2_aux b n c = (* n > 0 *)
  exp2_aux b (n-1) (c*b)

fun exp2 b n = exp2_aux b n 1

```

There is still a disadvantage to this implementation in that it carries out  $n$  multiplications. There is a yet more efficient implementation which

carries out only  $O(\log(n))$  multiplications by taking advantage of the observation that  $b^{2n} = (b^2)^n$ . That is, we can calculate  $b^{2n}$  by instead calculating  $b_2^n$  for a different base  $b_2$ . The corresponding inductive proof has a somewhat different structure from the proofs so far, because the step foreshadowed above reduces computing  $b^{2n}$  to computing  $b_2^n$ , which means given an (even)  $n > 0$ , we have to apply the induction hypothesis to  $n/2$ . A similar reasoning will apply for odd numbers. Fortunately,  $n/2 < n$  for  $n > 0$ , so the principle of *complete induction* allows this pattern of reasoning.

The statement of the lemma itself remains unchanged, only its proof.

**Lemma 5**  $\forall b:\text{nat. } \forall n:\text{nat. } \forall c:\text{nat. } \exists y:\text{nat. } y = c \times b^n$

**Proof:** By complete induction on  $n$ .

**Case:**  $n = 0$ . Then, as before, pick  $y = c$ .

**Case:**  $n > 0$ . Then we distinguish two cases:  $n$  is even or  $n$  is odd. Presumably this can be decided in our theory of natural numbers.

**Subcase:**  $n = 2k$  for some  $k < n$ . We have to prove  $\exists y:\text{nat. } y = c_1 \times b_1^n$  for some arbitrary  $c_1$  and  $b_1$  (by  $\forall I$ ).

$$\begin{aligned} \exists y:\text{nat. } y &= c_1 \times (b_1 \times b_1)^k && \text{by ind. hyp. for } b = b_1 \times b_1, n = k, \text{ and } c = c_1 \\ a &= c_1 \times (b_1 \times b_1)^k && \text{for some } a, \text{ by } \exists E \\ \text{Pick } y &= a = c_1 \times (b_1^2)^k = c_1 \times b_1^{2k} = c_1 \times b_1^n \\ \exists y:\text{nat. } y &= c_1 \times b_1^n && \text{by } \exists I \end{aligned}$$

**Subcase:**  $n = 2k + 1$  for some  $k < n$ . We have to prove  $\exists y:\text{nat. } y = c_1 \times b_1^n$  for some arbitrary  $c_1$  and  $b_1$  (by  $\forall I$ ).

$$\begin{aligned} \exists y:\text{nat. } y &= (c_1 \times b_1) \times (b_1 \times b_1)^k && \text{by ind. hyp. for } b = b_1 \times b_1, n = k, \text{ and } c = c_1 \times b \\ a &= (c_1 \times b_1) \times (b_1 \times b_1)^k && \text{for some } a, \text{ by } \exists E \\ \text{Pick } y &= a = (c_1 \times b_1) \times (b_1^2)^k = c_1 \times b_1^{2k+1} = c_1 \times b_1^n \\ \exists y:\text{nat. } y &= c_1 \times b_1^n && \text{by } \exists I \end{aligned}$$

□

The proof of the theorem does not change, but the extracted function now calls upon a different version of the auxiliary function because we have given a different proof. Note that it is still tail recursive, and we were able to put the accumulator to good use in the case of an odd number.

```

fun exp3_aux b 0 c = c
| exp3_aux b n c = (* n > 0 *)
  if n mod 2 = 0
  then exp3_aux (b*b) (n div 2) c
  else exp3_aux (b*b) ((n-1) div 2) (c*b)

fun exp3 b n = exp3_aux b n 1

```

## 4 Example: Warshall's Algorithm for Graph Reachability

This example is even less formal and sketchier than the previous section. It concerns induction about structures other than natural numbers, particularly lists.

To start with, how do we even specify graph reachability? We assume we are given a graph  $G$  via a type of nodes  $N$  and a collection of edges  $E$  connecting nodes. We consider the graph  $G$  fixed, so we won't repeatedly mention it in every proposition in the rest of this section.

We also have a notion of a *path* through a graph, following the set of edges. We write  $\text{path}(x, y)$  when there is a path  $p$  in the graph connecting  $x$  and  $y$ .<sup>1</sup> We can then specify graph reachability as

$$\forall x:N. \forall y:N. \text{path}(x, y) \vee \neg\text{path}(x, y)$$

Classically, this is a completely obvious statement, because it has the form  $\forall x. \forall y. A \vee \neg A$  which is trivially true. Constructively, a proof will have to show whether, given an  $x$  and  $y$ , there is a path connecting them or not. In addition, the proof of  $\text{path}(x, y)$  should exhibit such a path, while a proof of  $\neg\text{path}(x, y)$  should derive a contraction from the assumption that there is one. As in the previous examples, we will ignore some of the computational of the content of the proof, focusing on returning the boolean true if there is a path and false if there is none. In a later lecture we will see how we can systematically and formally hide some of the computational contents of proofs while keeping other information.

Now the statement above could be proved in a number of ways. For example, we might proceed by induction over the length of the potential path, or by induction over the number of unvisited nodes in the graph.

---

<sup>1</sup>In lecture, we used a slightly different version which made the path explicit, but that is not necessary to understand the basic idea.

Here, will use a different idea: consider a fixed enumeration of the vertices in the graph (a list of vertices) and proceed by induction over the structure of this list. Given some list  $V$  of vertices, we write  $\text{path}_V(x, y)$  if there is a path  $p$  connecting  $x$  and  $y$  using only vertices from  $V$  as interior nodes. The path  $p$  must therefore start with  $x$ , finish with  $y$ , and all other vertices on  $p$  must be in  $V$ .

Now we mildly generalize our statement so we can prove it inductively:

$$\forall V: N \text{ list. } \forall x: N. \forall y: N. \text{path}_V(x, y) \vee \neg \text{path}_V(x, y)$$

Our original theorem follows easily by picking  $V = N$ , because then the path is allowed to contain all vertices.

**Theorem 6**  $\forall V: N \text{ list. } \forall x: N. \forall y: N. \text{path}_V(x, y) \vee \neg \text{path}_V(x, y)$

**Proof:** By induction on the structure of  $V$ .

**Base:**  $V = \text{nil}$ , the empty list. Then  $\text{path}_{\text{nil}}(x, y)$  exactly if there is a direct edge from  $x$  to  $y$  because no interior nodes are allowed.

**Step:**  $V = z :: W$  for some vertex  $z$  and remaining list  $W$ . By induction hypothesis,  $\text{path}_W(x, y) \vee \neg \text{path}_W(x, y)$ . We distinguish the two cases:

**Case:**  $\text{path}_W(x, y)$ . Then also  $\text{path}_{z::W}(x, y)$  since we do not even need to use  $z$ .

**Case:**  $\neg \text{path}_W(x, y)$ . Now we use the induction hypothesis again on  $W$ , but this time on  $x$  and  $z$ , so  $\text{path}_W(x, z) \vee \neg \text{path}_W(x, z)$ . We once again distinguish two cases:

**Subcase:**  $\text{path}_W(x, z)$ . Now we use the induction hypothesis a third time to see if there is a path from  $z$  to  $y$  using only  $W$ :  $\text{path}_W(z, y) \vee \neg \text{path}_W(z, y)$ . Again, we distinguish these cases:

**Subsubcase:**  $\text{path}_W(z, y)$ . Since also  $\text{path}_W(x, z)$  we can concatenate these two paths to obtain  $\text{path}_{z::W}(x, y)$ .  $z$  has to be added, because it is on the interior of the path that goes from  $x$  to  $y$ , but that's fine since  $V = z :: W$ .

**Subsubcase:**  $\neg \text{path}_W(z, y)$ . Then  $\neg \text{path}_{z::W}(x, y)$ : if there is no path from  $x$  to  $y$  entirely over  $W$ , allowing  $z$  does not help if there is no path from  $z$  to  $y$  over  $W$ .

**Subcase:**  $\neg \text{path}_W(x, z)$ . Then also  $\neg \text{path}_{z::W}(x, y)$ .

□

In writing out the computational content we replace the if-then-else constructs with corresponding uses of `orelse` and `andalso`, for the sake of brevity and readability. So we expand as follows:

```
b orelse c == if b then true else c
b andalso c == if b then c else false
```

The code then turns out to be exceedingly compact.

```
fun warshall edge (nil) x y = edge x y
| warshall edge (z::W) x y =
  warshall edge W x y orelse
  (warshall edge W x z andalso warshall edge W z y)
```

The compiler tells us

```
val warshall = fn : ('a -> 'a -> bool) -> 'a list -> 'a -> 'a -> bool
```

which means that this works for any type of vertices '`a` as long as we have a function representing the edge relation.

This does not quite capture Warshall's algorithm yet: to get the right complexity we need to represent the *function* `warshall edge`  $V$  as a two-dimensional *boolean array* indexed by  $x$  and  $y$ , and for each  $z$  run through all pairs  $x$  and  $y$  to fill it with booleans.<sup>2</sup> This transformation can be carried out informally, or rigorously as shown in [Pfe90]. Of course, for small graphs the SML source code including a small example<sup>3</sup> works just fine.

If now look back at the proof we can see that it actually contains enough information to also extract the path when there is one. If a path is represented by a list of vertices, the result of of an extracted function which return the path if there is one will have type

```
val warshall2 : ('a -> 'a -> bool) -> 'a list
              -> 'a -> 'a -> 'a list option
```

which is implemented by the following function

---

<sup>2</sup>See, for example, the Wikipedia page on then [Floyd-Warshall algorithm](#)

<sup>3</sup><http://www.cs.cmu.edu/fp/courses/15317-f17/lectures/07-induction.sml>

```

fun warshall2 edge (nil) x y =
  if edge x y then SOME [x,y] else NONE
| warshall2 edge (z::W) x y =
  case warshall2 edge W x y
    of SOME p => SOME p
    | NONE => (case warshall2 edge W x z
      of SOME q => (case warshall2 edge W z y
        of SOME r => SOME (q @ tl r)
        | NONE => NONE)
      | NONE => NONE)

```

## 5 Bonus Example: Tail-Recursive Integer Square Root

We did not have time to discuss this in lecture, but we may consider how we can make the integer square root example more efficient. In particular, we should see if we can make it tail recursive. The key idea is the same that might occur to anyone when ask the implement integer square root: we add a counter  $c$  which we increment until  $c^2$  exceeds  $x$ . The problem now becomes how to state the theorem and how to find a corresponding proof.

The counter needs to become a new argument of an auxiliary function, so in the lemma there will be additional quantifier. All the quantifiers range over natural numbers, so we omit the type. We try

$$\forall x. \forall c. c^2 \leq x \supset \exists y. y^2 \leq x \wedge x < (y + 1)^2$$

At first sight this might look wrong since  $c$  does not occur in the scope of the quantifier on  $y$ , but the information about  $c$  may help us to construct such a  $y$  anyway.

In the proof, what becomes smaller as we count  $c$  upward? Clearly, it is the distance between  $c$  and  $x$  or, more precisely, the distance between  $c^2$  and  $x$ . When this distance becomes 0, we terminate the recursion. This leads to the following lemma, proof, and theorem:

**Lemma 7**  $\forall x. \forall c. c^2 \leq x \supset \exists y. y^2 \leq x \wedge x \leq (y + 1)^2$

**Proof:** By complete induction on  $x - c^2$ .

We have to prove, for an arbitrary  $x$  and  $c$  with  $c^2 \leq x$  that  $\exists y. y^2 \leq x \wedge x < (y + 1)^2$ . We distinguish two cases:

**Case:**  $x < (c + 1)^2$ . Then we can pick  $y = c$  since  $c^2 \leq x$  (by assumption) and  $x < (c + 1)^2$  (this case).

**Case:**  $(c + 1)^2 \leq x$ . Then we can apply the induction hypothesis, precisely because  $(c + 1)^2 \leq x$  and  $x - (c + 1)^2 = x - c^2 - 2c - 1 < x - c^2$ .

$$\exists y. y^2 \leq x \wedge x < (y + 1)^2 \quad \text{by ind. hyp.}$$

But this is exactly what we needed to prove.

□

**Theorem 8**  $\forall x. \exists y. y^2 \leq x \wedge x \leq (y + 1)^2$

**Proof:** Use the lemma for  $c = 0$ , which satisfies the requirement because  $c^2 = 0 \leq x$  for any  $x$ . □

The extracted function then looks as follows:

```
fun isqrt2_aux x c = (* c*c <= x *)
  if x < (c+1)*(c+1)
  then c
  else (* (c+1)*(c+1) <= x *)
    isqrt2_aux x (c+1)

fun isqrt2 x = isqrt2_aux x 0
```

We can take the analysis a bit further and try to ask: what does an induction over  $x - c^2$  actually mean? One possible interpretation is to add another variable  $d$  and constrain it to be *equal* to  $x - c^2$  so we apply complete induction on this variable.

**Lemma 9**  $\forall x. \forall d. \forall c. c^2 \leq x \wedge d = x - c^2 \supset \exists y. y^2 \leq x \wedge x \leq (y + 1)^2$

**Proof:** Proof is by complete induction on  $d$ . Assume we have an  $x, d$ , and  $c$  such that  $c^2 \leq x$  and  $d = x - c^2$ .

Previously, we distinguished cases based on whether  $x < (c + 1)^2$  or not. But we can rephrase test in terms of  $d$ :  $x < c^2 + 2c + 1$  iff  $x - c^2 < 2c + 1$  iff  $d < 2c + 1$ .

**Case:**  $d < 2c + 1$ . Then  $y = c$  satisfies the theorem because also  $c^2 \leq x$  by assumption and  $x < (c + 1)^2$  in this case.

**Case:**  $d \geq 2c + 1$ . Then  $(c + 1)^2 \leq x$  and  $0 \leq x - (c + 1)^2 = x - c^2 - 2c - 1 = d - 2c - 1 < d$  so we can apply the induction hypothesis on  $d - 2c - 1$  and  $c + 1$  to obtain some  $y$  such that  $y^2 \leq x \wedge x \leq (y + 1)^2$ .

□

**Theorem 10**  $\forall x. \exists y. y^2 \leq x \wedge x \leq (y + 1)^2$

**Proof:** Use the lemma for  $d = x$  and  $c = 0$ , which satisfy the requirements because  $c^2 = 0 \leq x$  for any  $x$  and  $x - c^2 = x$  □

The code, leaving out any extraneous proof information:

```
fun isqrt3_aux x d c =
  if d < 2*c+1
  then c
  else isqrt3_aux x (d-2*c-1) (c+1)

fun isqrt3 x = isqrt3_aux x x 0
```

Note that this remaind tail recursive and avoids the potentially “costly” multiplication  $(c + 1) \times (c + 1)$  on every recursive call from the previous version.

## References

- [Pfe90] Frank Pfenning. Program development through proof transformation. *Contemporary Mathematics*, 106:251–262, 1990.

# Lecture Notes on Heyting Arithmetic

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 8  
September 21, 2017

## 1 Introduction

In this lecture we discuss the data type of natural numbers. They serve as a prototype for a variety of inductively defined data types, such as lists or trees. Together with quantification previous lecture, this allow us to reason constructively about natural numbers and extract corresponding functions. The constructive system for reasoning logically about natural numbers is called *intuitionistic arithmetic* or *Heyting arithmetic* [Hey56]. The *classical* version of the same principles is called *Peano arithmetic* [Pea89]. Both of these are usually introduced *axiomatically* rather than as an extension of natural deduction as we do here.

## 2 Induction

As usual, we think of the type of natural numbers as defined by its introduction form. Note, however, that *nat* is a *type* rather than a proposition. It is possible to completely unify these concepts to arrive at *type theory*, something we might explore later in this course. For now, we just specify cases for the typing judgment  $t : \tau$ , read term  $t$  has type  $\tau$ , that was introduced in the previous lecture on quantification, but for which we have seen no specific instances yet. We distinguish this from  $M : A$  which has the same syntax, but relates a proof term to a proposition instead of a term to a type.

---

\*Edits by André Platzer.

There are two introduction rules, one for zero and one for successor.

$$\frac{}{0 : \text{nat}} \text{nat}I_0 \quad \frac{n : \text{nat}}{\text{s } n : \text{nat}} \text{nat}I_s$$

Intuitively, these rules express that 0 is a natural number ( $\text{nat}I_0$ ) and that the successor  $\text{s } n$  is a natural number if  $n$  is a natural number. This definition has a different character from the previous definitions. For example, we defined the meaning of  $A \wedge B$  *true* from the meanings of  $A$  *true* and the meaning of  $B$  *true*, all of which are propositions. It is even different from the proof term assignment rules where, for example, we defined  $\langle M, N \rangle : A \wedge B$  in terms of  $M : A$  and  $N : B$ . In each case, the proposition is decomposed into its parts.

Here, the types in the conclusion and premise of the  $\text{nat}I_s$  rules are the same, namely  $\text{nat}$ . Fortunately, the *term*  $n$  in the premise is a part of the term  $\text{s } n$  in the conclusion, so the definition is not circular, because the judgment in the premise is still smaller than the judgment in the conclusion. In (verificationist) constructive logic truth is defined by the introduction rules. The resulting implicit principle, that nothing is true unless the introduction rules prove it to be true, is of deep significance here. Nothing else is a natural number, except the objects constructed via  $\text{nat}I_s$  from  $\text{nat}I_0$ . The rational number  $\frac{7}{4}$  cannot sneak in claiming to be a natural number (which, by  $\text{nat}I_s$  would also make its successor  $\frac{11}{4}$  claim to be natural).

But what should the elimination rule be? We cannot decompose the proposition into its parts, so we decompose the term instead. Natural numbers have two introduction rules just like disjunctions. Their elimination rule, thus, also proceeds by cases, accounting for the possibility that a given  $n$  of type  $\text{nat}$  is either 0 or  $\text{s } x$  for some  $x$ . A property  $C(n)$  is true if it holds no matter whether the natural number  $n$  was introduced by  $\text{nat}I_0$  so is zero or was introduced by  $\text{nat}I_s$  so is a successor.

$$\frac{\begin{array}{c} \frac{}{x : \text{nat}} \quad \frac{\begin{array}{c} C(x) \text{ true} \\ \vdots \\ C(\text{s } x) \text{ true} \end{array}}{C(x) \text{ true}} u \\ \hline \text{nat}E^{x,u} \end{array}}{C(n) \text{ true}}$$

In words: In order to prove property  $C$  of a natural number  $n$  we have to prove  $C(0)$  and also  $C(\text{s } x)$  under the assumption that  $C(x)$  for a new parameter  $x$ . The scope of  $x$  and  $u$  is just the rightmost premise of the rule.

This corresponds exactly to proof by induction, where the proof of  $C(0)$  is the base case, and the proof of  $C(\mathbf{s} x)$  from the assumption  $C(x)$  is the induction step. That is why  $\text{nat}E^{x,u}$  is also called an *induction rule* for  $\text{nat}$ .

We managed to state this rule without any explicit appeal to universal quantification, using parametric judgments instead. We could, however, write it down with explicit quantification, in which case it becomes:

$$\forall n:\text{nat}. \ C(0) \supset (\forall x:\text{nat}. \ C(x) \supset C(\mathbf{s} x)) \supset C(n)$$

for an arbitrary property  $C$  of natural numbers. It is an easy exercise to prove this with the induction rule above, since the respective introduction rules lead to a proof that exactly has the shape of  $\text{nat}E^{x,u}$ .

**All natural numbers are zero or successors.** To illustrate this rule in action, we start with a very simple property: every natural number is either 0 or has a predecessor. First, a detailed induction proof in the usual mathematical style and then a similar formal proof.

**Theorem:**  $\forall x:\text{nat}. \ x = 0 \vee \exists y:\text{nat}. \ x = \mathbf{s} y$ .

**Proof:** By induction on  $x$ .

**Case:**  $x = 0$ . Then the left disjunct is true.

**Case:**  $x = \mathbf{s} x'$ . Then the right disjunct is true: pick  $y = x'$  and observe  $x = \mathbf{s} x' = \mathbf{s} y$ .

Next we write this in the formal notation of inference rules. We suggest the reader try to construct this proof step-by-step; we show only the final deduction. We assume there is either a primitive or derived rule of inference called  $\text{refl}$  expressing reflexivity of equality on natural numbers ( $n = n$ ). We use the same names as in the mathematical proof.

$$\frac{}{x : \text{nat}} \frac{}{\overline{0 = 0 \text{ true}} \text{ refl}} \frac{}{\overline{0 = 0 \vee \exists y:\text{nat}. \ 0 = \mathbf{s} y \text{ true}} \vee I_1} \quad \frac{\overline{x' : \text{nat}} \quad \overline{\mathbf{s} x' = \mathbf{s} x' \text{ true}} \text{ refl}}{\exists y:\text{nat}. \ \mathbf{s} x' = \mathbf{s} y \text{ true}} \exists I \quad \frac{\overline{\mathbf{s} x' = 0 \vee \exists y:\text{nat}. \ \mathbf{s} x' = \mathbf{s} y \text{ true}} \vee I_2}{\overline{x = 0 \vee \exists y:\text{nat}. \ x = \mathbf{s} y \text{ true}} \text{ nat}E^{x,u}}$$

$$\frac{x = 0 \vee \exists y:\text{nat}. \ x = \mathbf{s} y \text{ true}}{\forall x:\text{nat}. \ x = 0 \vee \exists y:\text{nat}. \ x = \mathbf{s} y \text{ true}} \forall I^x$$

This is a simple proof by cases and, in this particular proof, does not even use the induction hypothesis  $x' = 0 \vee \exists y:\text{nat}. \ x' = \mathbf{s} y \text{ true}$ , which would

have been labeled  $u$ . It is also possible to finish the proof by eliminating from that induction hypothesis, but the proof then ends up being more complicated. At our present level of understanding, the computational counterpart for the above proof might be a zero-check function for natural numbers. It takes any natural number and provides the left disjunct if that number was 0 while providing the right disjunct if it was a successor. Making use of the witness, we will later discover more general computational content once we have a proof term assignment.

In the application of the induction rule  $\text{nat}E$  we used the property  $C(x)$ , which is a proposition with the free variable  $x$  of type  $\text{nat}$ . To write it out explicitly:

$$C(x) = (x = 0 \vee \exists y: \text{nat}. x = s y)$$

While getting familiar with formal induction proofs it may be a good idea to write out the induction formula explicitly.

### 3 Equality

We already used equality in the previous example, without justification, so we now introduce it into our formal system. It is certainly a central part of Heyting (and Peano) arithmetic.

There are many ways to define and reason with equality. The one we choose here is the one embedded in arithmetic where we are only concerned with numbers. Thus we are trying to define  $x = y$  only for natural numbers  $x$  and  $y$ . Of course,  $x = y$  must be a *proposition*, not a term. As a proposition, we will use the techniques of the course and define it by means of introduction and elimination rules!

The introduction rules are straightforward.<sup>1</sup>

$$\frac{}{0 = 0 \text{ true}} =I_{00} \quad \frac{x = y \text{ true}}{s x = s y \text{ true}} =I_{ss}$$

If we take this as our definition of equality on natural numbers, how can we use the knowledge that  $n = k$ ? If  $n$  and  $k$  are both zero, we cannot learn anything. If both are successors, we know their argument must be equal. Finally, if one is a successor and the other zero, then this is contradictory

---

<sup>1</sup>As a student observed in lecture, we could also just state  $x = x \text{ true}$  as an inference rule with no premise. However, it is difficult to justify the elimination rules we need for Heyting arithmetic from this definition.

and we can derive anything.

$$\frac{\text{no rule } E_{00}}{0 = s x \text{ true}} = E_{0s} \quad \frac{s x = 0 \text{ true}}{C \text{ true}} = E_{s0} \quad \frac{s x = s y \text{ true}}{x = y \text{ true}} = E_{ss}$$

Local soundness is very easy to check, but what about local completeness? It turns out to be a complicated issue so we will not discuss it here.

## 4 Equality is Reflexive

As a simple inductive theorem we now present the reflexivity of equality.

**Theorem 1**  $\forall x:\text{nat}. x = x$

**Proof:** By induction on  $x$ .

**Base:**  $x = 0$ . Then  $0 = 0$  by rule  $=I_{00}$

**Step:** Assume  $x = x$ . We have to show  $s x = s x$ , which follows by  $=I_{ss}$ .

□

This proof is small enough so we can present it in the form of a natural deduction. For induction, we use  $C(n) = (n = n)$ .

$$\frac{n : \text{nat} \quad \frac{}{0 = 0 \text{ true}} = I_{00} \quad \frac{\overline{x = x \text{ true}}^u}{s x = s x \text{ true}} = I_{ss}}{\frac{n = n \text{ true}}{\forall x:\text{nat}. x = x \text{ true}}} \text{ nat}E^{x,u}$$

The hypothesis  $x : \text{nat}$  introduced by  $\text{nat}E^{x,u}$  is implicitly used to establish that  $x = x$  is a well-formed proposition, but is otherwise not explicit in the proof.

Now we can define a derived rule of inference:

$$\frac{x : \text{nat}}{x = x \text{ true}} \text{ refl}$$

by using  $\forall E$  with the theorem just proved. We usually suppress the premise  $x : \text{nat}$  since we already must know  $x : \text{nat}$  for the proposition  $x = x$  to be well-formed. This is the rule we used in Section 2.

## 5 Primitive Recursion

Reconsidering the elimination rule for natural numbers, we can notice that we exploit the knowledge that  $n : \text{nat}$ , but we only do so when we are trying to establish the truth of a proposition,  $C(n)$ . However, we are equally justified in using  $n : \text{nat}$  when we are trying to establish a typing judgment of the form  $t : \tau$ . The rule, also called *rule of primitive recursion* for  $\text{nat}$ , then becomes

$$\frac{\begin{array}{c} \overline{x : \text{nat}} \quad \overline{r : \tau} \\ \vdots \\ n : \text{nat} \quad t_0 : \tau \quad t_s : \tau \end{array}}{R(n, t_0, x. r. t_s) : \tau} \text{nat}E^{x,r}$$

Here,  $R$  is a new term constructor,<sup>2</sup> the term  $t_0$  is the zero case where  $n = 0$ , and the term  $t_s$  captures the successor case where  $n = s n'$ . In the latter case  $x$  is a new parameter introduced in the rule that stands for  $n'$ . And  $r$  stands for the result of the function  $R$  when applied to  $n'$ , which corresponds to an appeal to the induction hypothesis. The notation  $x. r. t_s$  indicates that occurrences of  $x$  and  $r$  in  $t_s$  are bound with scope  $t_s$ . The fact that both are bound corresponds to the assumptions  $x : \text{nat}$  and  $r : \tau$  that are introduced to prove  $t_s : \tau$  in the rightmost premise.

The local reduction rules may help explain this. We first write them down just on the terms, where they are computation rules.

$$\begin{aligned} R(0, t_0, x. r. t_s) &\xrightarrow{R} t_0 \\ R(s n', t_0, x. r. t_s) &\xrightarrow{R} [R(n', t_0, x. r. t_s)/r][n'/x] t_s \end{aligned}$$

The second case reduces to the term  $t_s$  with parameter  $x$  instantiated to the number  $n'$  of the inductive hypothesis and parameter  $r$  instantiated to the value of  $R$  at  $n'$ . So the argument  $t_0$  of  $R$  indicates the output to use for  $n = 0$  and  $t_s$  indicates the output to use for  $n = s x$  as a function of the smaller number  $x$  and of  $r$  for the recursive outcome of  $R(n, t_0, x. r. t_s)$ .

These are still quite unwieldy, so we consider a more readable schematic form, called the *schema of primitive recursion*. If we define  $f$  by cases

$$\begin{aligned} f(0) &= t_0 \\ f(s x) &= t_s(x, f(x)) \end{aligned}$$

---

<sup>2</sup> $R$  suggests recursion

where the only occurrence of  $f$  on the right-hand side is applied to  $x$ , then we could have defined  $f$  explicitly with

$$f = \text{fn } n \Rightarrow R(n, t_0, x. r. t_s(x, r))$$

To verify this, apply  $f$  to 0 and apply the reduction rules and also apply  $f$  to  $s n$  for an arbitrary  $n$  and once again apply the reduction rules.

$$\begin{aligned} f(0) &\xrightarrow{R} R(0, t_0, x. r. t_s(x, r)) \\ &\xrightarrow{R} t_0 \end{aligned}$$

noting that the  $x$  in  $x.r.t_s(\dots)$  is not a free occurrence (indicated by the presence of the dot in  $x$ ) since it corresponds to the hypothesis  $x : \text{nat}$  in  $\text{nat}E^{x,r}$ . Finally

$$\begin{aligned} f(s n) &\xrightarrow{R} R(s n, t_0, x. r. t_s(x, r)) \\ &\xrightarrow{R} t_s(n, R(n, t_0, x. r. t_s(x, r))) \\ &= t_s(n, f(n)) \end{aligned}$$

The last equality is justified by a (meta-level) induction hypothesis, because we are trying to show that  $f(n) = R(n, t_0, x. r. t_s(x, r))$

We also reuse the notion of functional abstraction (already used to describe proof terms of  $A \supset B$  and  $\forall x:\tau. A(x)$  to describe functions at the level of data.<sup>3</sup> We write  $\tau \rightarrow \sigma$  for types  $\tau$  and  $\sigma$  and present them here without further justification since they just mirror the kinds of rules we have seen multiple times already.

$$\frac{\overline{x : \tau} \quad \vdots \quad s : \sigma}{\lambda x:\tau. s : \tau \rightarrow \sigma} \rightarrow I \qquad \frac{s : \tau \rightarrow \sigma \quad t : \tau}{s t : \sigma} \rightarrow E$$

The local reduction is

$$(\lambda x:\tau. s) t \xrightarrow{R} [t/x]s$$

Now we can define double via the schema of primitive recursion.

$$\begin{aligned} \text{double}(0) &= 0 \\ \text{double}(s x) &= s(s(\text{double } x)) \end{aligned}$$

---

<sup>3</sup>The case for unifying all these notions in type theory looks pretty strong at this point.

We can read off the closed-form definition if we wish:

$$\text{double} = \lambda n. R(n, 0, x. r. s(s r))$$

After having understood this, we will be content with using the schema of primitive recursion. We define addition and multiplication as exercises.

$$\begin{aligned}\text{plus}(0) &= \lambda y. y \\ \text{plus}(s x) &= \lambda y. s((\text{plus } x) y)\end{aligned}$$

Notice that plus is a function of type  $\text{nat} \rightarrow (\text{nat} \rightarrow \text{nat})$  that is primitive recursive in its (first and only) argument.

$$\begin{aligned}\text{times}(0) &= \lambda y. 0 \\ \text{times}(s x) &= \lambda y. (\text{plus } ((\text{times } x) y)) y\end{aligned}$$

## 6 Proof Terms

With proof terms for primitive recursion in place, we can revisit and make a consistent proof term assignment for the elimination form with respect to the truth of propositions.

$$\frac{\overline{x : \text{nat}} \quad \overline{u : C(x)}^u \quad \vdots}{\frac{n : \text{nat} \quad M_0 : C(0) \quad M_s : C(s x)}{R(n, M_0, x. u. M_s) : C(n)}} \text{nat}E^{x,u}$$

Except for the type of judgment (proof terms and propositions versus typing judgments), this elimination rule  $\text{nat}E^{x,u}$  is essentially the same as the (primitive) recursion rule  $\text{nat}E^{x,r}$ , just on propositions instead of data.

The local reductions we discussed before for terms representing data, also work for these proofs terms, because they are both derived from slightly different variants of the elimination rules (one with proof terms, one with data terms).

$$\begin{aligned}R(0, M_0, x. u. M_s) &\xrightarrow{\text{ }} M_0 \\ R(s n', M_0, x. u. M_s) &\xrightarrow{\text{ }} [R(n', M_0, x. u. M_s)/u][n'/x] M_s\end{aligned}$$

We can conclude that proofs by induction correspond to functions defined by primitive recursion, and that they compute in the same way.

Returning to the earlier example, we can now write the proof terms, using  $\_$  for proofs of equality (whose computational content we do not care about).

**Theorem:**  $\forall x:\text{nat}. x = 0 \vee \exists y:\text{nat}. x = s y$ .

**Proof:** By induction on  $x$ .

Case:  $x = 0$ . Then the left disjunct is true.

Case:  $x = s x'$ . Then the right disjunct is true: pick  $y = x'$  and observe  $x = s x' = s y$ .

The extracted function is the predecessor function:

$$\text{pred} = \lambda x:\text{nat}. R(x, \text{inl } \_, x. r. \text{inr}(x, \_))$$

here we have suppressed the evidence for equalities, since we have not yet introduced proof terms for them.

## 7 Local Proof Reduction

We would like to check that the rules for natural numbers are locally sound and complete. For soundness, we verify that no matter how we introduce the judgment  $n : \text{nat}$ , we can find a “more direct” proof of the conclusion. In the case of  $\text{natI}_0$  this is easy to see, because the second premise already establishes our conclusion directly.

$$\frac{\frac{\frac{}{x : \text{nat}} \quad \frac{}{C(x) \text{ true}} u}{\text{natI}_0} \quad \frac{\mathcal{E}}{C(0) \text{ true}} \quad \frac{\mathcal{F}}{C(s x) \text{ true}}}{\text{natE}^{x,u}} \implies_R \frac{}{C(0) \text{ true}} \quad C(0) \text{ true}}$$

The case where  $n = s n'$  is more difficult and more subtle. Intuitively, we should be using the deduction of the second premise for this case.

$$\begin{array}{c}
 \frac{\mathcal{D} \quad \frac{n' : \text{nat}}{\mathsf{nat}I_s} \quad \frac{\mathcal{E} \quad C(0) \text{ true}}{C(sx) \text{ true}} \quad \frac{x : \text{nat} \quad \frac{C(x) \text{ true}}{u}}{C(sx) \text{ true}}^u}{C(sn') \text{ true}} \quad \frac{\mathcal{F} \quad C(sx) \text{ true}}{\mathsf{nat}E^{x,u}} \\
 \hline
 \frac{\mathcal{D} \quad \frac{n' : \text{nat}}{\mathsf{nat}I_s} \quad \frac{\mathcal{E} \quad C(0) \text{ true}}{C(n') \text{ true}} \quad \frac{\mathcal{F} \quad C(sx) \text{ true}}{C(n') \text{ true}} \quad \frac{x : \text{nat} \quad \frac{C(x) \text{ true}}{u}}{C(n') \text{ true}}^u}{C(sn') \text{ true}} \\
 \hline
 \mathrel{\Rightarrow_R} \frac{}{[n'/x]\mathcal{F}' \quad C(sn') \text{ true}}
 \end{array}$$

It is difficult to see in which way this is a reduction:  $\mathcal{D}$  is duplicated,  $\mathcal{E}$  persists, and we still have an application of  $\mathsf{nat}E$ . The key is that the term we are eliminating with the applicaton of  $\mathsf{nat}E$  becomes smaller: from  $sn'$  to  $n'$ . In hindsight we should have expected this, because the term is also the only component getting smaller in the second introduction rule for natural numbers. Fortunately, the term that  $\mathsf{nat}E$  is applied to can only get smaller finitely often because it will ultimately just be 0, so will be back in the first local reduction case.

The computational content of this reduction is more easily seen in a different context, so we move on to discuss primitive recursion.

The question of local expansion does not make sense in our setting. The difficulty is that we need to show that we can apply the elimination rules in such a way that we can reconstitute a proof of the original judgment. However, the elimination rule we have so far works only for the truth judgment, so we cannot really reintroduce  $n : \text{nat}$ , since the only two introduction rules  $\mathsf{nat}I_0$  and  $\mathsf{nat}I_s$  do not apply.

## 8 Local Expansion

Using primitive recursion, we can now write a local expansion.

$$\frac{\mathcal{D} \quad n : \text{nat} \quad \mathrel{\Rightarrow_E} \frac{\mathcal{D} \quad \frac{n : \text{nat} \quad 0 : \text{nat}}{\mathsf{nat}I_0} \quad \frac{x : \text{nat} \quad \frac{sx : \text{nat}}{R(n, 0, x. r. sx) : \text{nat}}}{\mathsf{nat}I_s}}{\mathsf{nat}E^{x,r}}$$

A surprising observation about the local expansion is that it does not use the recursive result,  $r$ , which corresponds to a use of the induction hypothesis. Consequently, a simple proof-by-cases that uses  $\text{nat}E_0$  when  $n$  is zero and uses  $\text{nat}E_s$  when  $n$  is a successor would also have been locally sound and complete.

This is a reflection of the fact that the local completeness property we have does not carry over to a comparable global completeness. The difficulty is the well-known property that in order to prove a proposition  $A$  by induction, we may have to first generalize the induction hypothesis to some  $B$ , prove  $B$  by induction and also prove  $B \supset A$ . Such proofs do not have the subformula property, which means that our strict program of explaining the meaning of propositions from the meaning of their parts breaks down in arithmetic. In fact, there is a hierarchy of arithmetic theories, depending on which propositions we may use as induction formulas.

## References

- [Hey56] Arend Heyting. *Intuitionism: An Introduction*. North-Holland Publishing, Amsterdam, 1956. 3rd edition, 1971.
- [Pea89] Giuseppe Peano. *Arithmetices Principia, Nova Methodo Exposita*. Fratres Bocca, 1889.

# Lecture Notes on Sequent Calculus

15-317: Constructive Logic  
Frank Pfenning

Lecture 9  
October 3, 2017

## 1 Introduction

In this lecture we shift to a different presentation style for proof calculi. We develop the sequent calculus as a formal system for proof search in natural deduction. In addition to enabling an understanding of proof search, sequent calculus leads to a more transparent management of the scope of assumptions during a proof, and also allows us more proof theory, so proofs about properties of proofs.

Sequent calculus was originally introduced by Gentzen [Gen35], primarily as a technical device for proving consistency of predicate logic. Our goal of describing a proof search procedure for natural deduction predisposes us to a formulation due to Kleene [Kle52] called  $G_3$ .

Our sequent calculus is designed to *exactly* capture the notion of a *verification*, introduced in [Lecture 5](#). Recall that verifications are constructed bottom-up, from the conclusion to the premises using introduction rules, while uses are constructed top-down, from hypotheses to conclusions using elimination rules. They meet in the middle, where a proposition we have deduced from assumptions may be used as a verification. In the sequent calculus, both steps work bottom-up, which will ultimately allow us to prove global versions of the local soundness and completeness properties.

## 2 Sequents

When constructing a verification, we are generally in a state of the following form

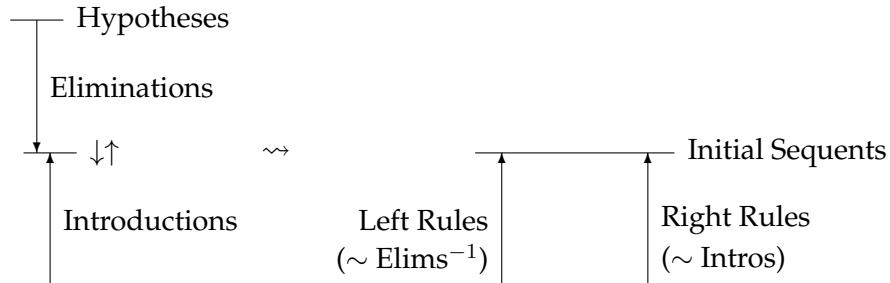
$$\begin{array}{c} A_1 \downarrow \quad \cdots \quad A_n \downarrow \\ \vdots \\ C \uparrow \end{array}$$

where  $A_1, \dots, A_n$  embody knowledge we may *use*, while  $C$  is the conclusion we are trying to *verify*. A *sequent* is just a local notation for such a partially complete verification. We write

$$A_1 \text{ left}, \dots, A_n \text{ left} \implies C \text{ right}$$

where the judgments  $A$  left and  $C$  right correspond to  $A \downarrow$  and  $C \uparrow$ , respectively. The judgments on the left are assumptions called *antecedents*, the judgment on the right is the conclusion we are trying to verify called the *succedent*. Sequent calculus is explicit about the assumptions that are available for use (antecedent) and about the proposition to be verified (succedent).

The rules that define the  $A$  left and  $A$  right judgment are systematically constructed from the introduction and elimination rules, keeping in mind their directions in terms of verifications and uses. Introduction rules are translated to corresponding *right rules*. Since introduction rules already work from the conclusion to the premises, this mapping is straightforward. Elimination rules work top-down, so they have to be flipped upside-down in order to work as sequent rules, and are turned into *left rules*. Pictorially:



We now proceed connective by connective, constructing the right and left rules from the introduction and elimination rules. When writing a sequent, we can always tell which propositions are on the left and which are on the right, so we omit the judgments left and right for brevity. Also, we abbreviate a collection of antecedents  $A_1 \text{ left}, \dots, A_n \text{ left}$  by  $\Gamma$ . The order

of the antecedents does not matter, so we will allow them to be implicitly reordered.

**Conjunction.** We recall the introduction rule first and show the corresponding right rule.

$$\frac{A \uparrow \quad B \uparrow}{A \wedge B \uparrow} \wedge I \quad \frac{\Gamma \Rightarrow A \quad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B} \wedge R$$

The only difference is that the antecedents  $\Gamma$  are made explicit. Both premises have the same antecedents, because any assumption can be used in both subdeductions.

There are two elimination rules, so we have two corresponding left rules.

$$\frac{A \wedge B \downarrow}{A \downarrow} \wedge E_1 \quad \frac{\Gamma, A \wedge B, A \Rightarrow C}{\Gamma, A \wedge B \Rightarrow C} \wedge L_1$$

$$\frac{A \wedge B \downarrow}{B \downarrow} \wedge E_2 \quad \frac{\Gamma, A \wedge B, B \Rightarrow C}{\Gamma, A \wedge B \Rightarrow C} \wedge L_2$$

We preserve the *principal formula*  $A \wedge B$  of the left rule in the premise. This is because we are trying to model proof construction in natural deduction where assumptions can be used multiple times. If we temporarily ignore the copy of  $A \wedge B$  in the premise, it is easier to see how the rules correspond.

**Truth.** Truth is defined just by an introduction rule and has no elimination rule. Consequently, there is only a right rule in the sequent calculus and no left rule.

$$\frac{}{\top \uparrow} \top I \quad \frac{}{\Gamma \Rightarrow \top} \top R$$

**Implication.** Again, the right rule for implication is quite straightforward, because it models the introduction rule directly.

$$\frac{\overline{A \downarrow} \quad u}{B \uparrow} \supset^u \quad \frac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \supset B} \supset R$$

We see here one advantage of the sequent calculus over natural deduction: the scoping for additional assumptions is simple. The new antecedent  $A$  left is available anywhere in the deduction of the premise, because in the sequent calculus we only work bottom-up. Moreover, we arrange all the rules so that antecedents are *persistent*: they are always propagated from the conclusion to all premises.

The elimination rule is trickier, because it involves a more complicated combination of verifications and uses.

$$\frac{A \supset B \downarrow \quad A \uparrow}{B \downarrow} \supset E \quad \frac{\Gamma, A \supset B \Rightarrow A \quad \Gamma, A \supset B, B \Rightarrow C}{\Gamma, A \supset B \Rightarrow C} \supset L$$

In words: in order to use  $A \supset B$  to verify  $C$  we have to produce a verification of  $A$ , in which case we can then use  $B$  in the verification of  $C$ . The antecedent  $A \supset B$  is carried over to both premises to maintain persistence. Note that the premises of the left rule are reversed, when compared to the elimination rule to indicate that we do not want to make the assumption  $B$  unless we have already established  $A$ .

In terms of provability, there is some redundancy in the  $\supset L$  rule. For example, once we know  $B$ , we no longer need  $A \supset B$ , because  $B$  is a stronger assumption. As stressed above, we try to maintain the correspondence to natural deductions and postpone these kinds of optimization until later.

**Disjunction.** The right rules correspond directly to the introduction rules, as usual.

$$\frac{A \uparrow}{A \vee B \uparrow} \vee I_1 \quad \frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B} \vee R_1$$

$$\frac{B \uparrow}{A \vee B \uparrow} \vee I_2 \quad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \vee B} \vee R_2$$

The disjunction elimination rule was somewhat odd, because it introduced two new assumptions, one for each case of the disjunction. The left rule for disjunction actually has a simpler form that is more consistent with all the other rules we have shown so far.

$$\frac{\overline{A \downarrow} \quad u \quad \overline{B \downarrow} \quad w}{\begin{array}{c} \vdots \\ C \uparrow \end{array}} \vee E^{u,w} \quad \frac{\Gamma, A \vee B, A \Rightarrow C \quad \Gamma, A \vee B, B \Rightarrow C}{\Gamma, A \vee B \Rightarrow C} \vee L$$

As for implication, scoping issues are more explicit and simplified because the new assumptions  $A$  and  $B$  in the first and second premise, respectively, are available anywhere in the deduction above. But the assumption  $A$  is only available in the deduction for the left premise, while  $B$  is only available in the right premise. Sequent calculus is explicit about that. The sequent calculus formulation also makes it more transparent what the appropriate verification/uses assignment is.

**Falsehood.** Falsehood has no introduction rule, and therefore no right rule in the sequent calculus. To arrive at the left rule, we need to pay attention to the distinction between uses and verifications, or we can construct the 0-ary case of disjunction from above.

$$\frac{\perp \downarrow}{C \uparrow} \perp E \quad \frac{}{\Gamma, \perp \Rightarrow C} \perp L$$

**Completing verifications.** Recall that we cannot use an introduction rule to verify atomic propositions  $P$  because they cannot be broken down further. The only possible verification of  $P$  is directly via a use of  $P$ . In the version of verifications we have presented, we can complete the construction of a verification whenever  $A \downarrow$  is available to conclude  $A \uparrow$ .<sup>1</sup> This turns into a so-called *initial sequent* or application of the *identity rule*.

$$\frac{A \downarrow}{A \uparrow} \downarrow\uparrow \quad \frac{}{\Gamma, A \Rightarrow A} \text{id}$$

This rule has a special status in that it does not break down any proposition, but establishes a connection between two judgments. In natural deduction, it is the connection between uses and verifications; in sequent calculus, it is the connection between the left and right judgments.

As a simple example, we consider the proof of  $(A \vee B) \supset (B \vee A)$ .

$$\frac{\frac{\frac{\frac{\frac{\frac{}{A \vee B, A \Rightarrow A} \text{id}}{\frac{}{A \vee B, A \Rightarrow B \vee A} \vee R_2} \text{id}}{\frac{\frac{\frac{}{A \vee B, B \Rightarrow B} \text{id}}{\frac{\frac{\frac{}{A \vee B, B \Rightarrow B \vee A} \vee R_1}{\frac{\frac{\frac{}{A \vee B \Rightarrow B \vee A} \vee L}{\frac{\frac{}{(A \vee B) \supset (B \vee A)} \supset R}{\frac{}{\Rightarrow (A \vee B) \supset (B \vee A)}} \supset R}} \supset R}} \supset R}} \supset R}} \supset R}} \supset R$$

<sup>1</sup>There are stricter versions of this, where the  $\downarrow\uparrow$  rule can only be used for atomic propositions  $P$ . We will return to this point in the next lectures.

Observe that sequent calculus proofs are always constructed bottom-up, with the desired conclusion at the bottom, working upwards using the respective left or right proof rules in the antecedent or succedent.

### 3 Observations on Sequent Proofs

We have already mentioned that antecedents in sequent proofs are *persistent*: once an assumption is made, it is henceforth usable above the inference that introduces it. Sequent proofs also obey the important *subformula property*: if we examine the complete or partial proof above a sequent, we observe that all sequents are made up of subformulas of the sequent itself. This is consistent with the design criteria for the verifications: the verification of a proposition  $A$  may only contain subformulas of  $A$ . This is important from multiple perspectives. Foundationally, we think of verifications as defining the meaning of the propositions, so a verification of a proposition should only depend on its constituents. For proof search, it means we do not have to try to resort to some unknown formula, but can concentrate on subformulas of our proof goal.

If we trust for the moment that a proposition  $A$  is true if and only if it has a deduction in the sequent calculus (as  $\Rightarrow A$ ), we can use the sequent calculus to formally prove that some proposition can *not* be true in general. For example, we can prove that intuitionistic logic is *consistent*.

**Theorem 1 (Consistency)** *It is not the case that  $\Rightarrow \perp$ .*

**Proof:** No left rule is applicable, since there is no antecedent. No right rule is applicable, because there is no right rule for falsehood. The identity rule is not applicable either. Therefore, there cannot be a proof of  $\Rightarrow \perp$ .  $\square$

**Theorem 2 (Disjunction Property)** *If  $\Rightarrow A \vee B$  then either  $\Rightarrow A$  or  $\Rightarrow B$ .*

**Proof:** No left rule is applicable, since there is no antecedent. The only right rules that are applicable are  $\vee R_1$  and  $\vee R_2$ . In the first case, we have  $\Rightarrow A$ , in the second  $\Rightarrow B$ .  $\square$

**Theorem 3 (Failure of Excluded Middle)** *It is not the case that  $\Rightarrow A \vee \neg A$  for arbitrary  $A$ .*

**Proof:** From the disjunction property, either  $\Rightarrow A$  or  $\Rightarrow \neg A$ . For the first sequent, no rule applies. For the second sequent, only  $\supset R$  applies and we would have to have a deduction of  $A \Rightarrow \perp$ . But for this sequent no rule applies.  $\square$

Of course, there are still specific formulas  $A$  for which  $\Rightarrow A \vee \neg A$  will be provable, such as  $\Rightarrow \top \vee \neg \top$  or  $\Rightarrow \perp \vee \neg \perp$ , but not generally for any  $A$ .

There are other simple observations that are important for some applications. The first is called *weakening*, which means that we can add an arbitrary proposition to a derivable sequent and get another derivable sequent with a proof that has the same structure.

**Theorem 4 (Weakening)** *If  $\Gamma \Rightarrow C$  then  $\Gamma, A \Rightarrow C$  with a structurally identical deduction.*

**Proof:** Add  $A$  to every sequent in the given deduction of  $\Gamma \Rightarrow C$ , but never use it. The result is a structurally identical deduction of  $\Gamma, A \Rightarrow C$ .  $\square$

**Theorem 5 (Contraction)** *If  $\Gamma, A, A \Rightarrow C$  then  $\Gamma, A \Rightarrow C$  with a structurally identical deduction.*

**Proof:** Pick one copy of  $A$ . Wherever the other copy of  $A$  is used in the given deduction, use the first copy of  $A$  instead. The result is a structurally identical deduction with one fewer copy of  $A$ .  $\square$

The proof of contraction actually exposes an imprecision in our presentation of the sequent calculus. When there are two occurrences of a proposition  $A$  among the antecedents, we have no way to distinguish which one is being used, either as the principal formula of a left rule or in an initial sequent. It would be more precise to label each antecedent with a unique label and then track labels in the inferences. We may make this precise at a later stage in this course; for now we assume that occurrences of antecedents can be tracked somehow so that the proof above, while not formal, is at least somewhat rigorous.

Now we can show that double negation elimination does not hold in general.<sup>2</sup>

**Theorem 6 (Failure of Double Negation Elimination)** *It is not the case that  $\Rightarrow \neg\neg A \supset A$  for arbitrary  $A$ .*

---

<sup>2</sup>This proof was not covered in lecture.

**Proof:** Assume we have the shortest proof of  $\Rightarrow \neg\neg A \supset A$ . There is only one rule that could have been applied ( $\supset R$ ), so we must also have a proof of  $\neg\neg A \Rightarrow A$ . Again, only one rule could have been applied,

$$\frac{\neg\neg A \Rightarrow \neg A \quad \neg\neg A, \perp \Rightarrow A}{\neg\neg A \Rightarrow A} \supset L$$

We can prove the second premise, but not the first. If such a proof existed, it must end either with the  $\supset R$  or  $\supset L$  rules, because these are the only applicable rules.

Case: The proof proceeds with  $\supset R$ .

$$\frac{\neg\neg A, A \Rightarrow \perp}{\neg\neg A \Rightarrow \neg A} \supset R$$

Now only  $\supset L$  could have been applied, and its premises must have been

$$\frac{\neg\neg A, A \Rightarrow \neg A \quad \neg\neg A, A, \perp \Rightarrow \perp}{\neg\neg A, A \Rightarrow \perp} \supset L$$

Again, the second premise could have been deduced, but not the first. If it had been inferred with  $\supset R$  and, due to contraction, we would end up with another proof of a sequent we have already seen, and similarly if  $\supset L$  had been used. In either case, it would contradict the assumption of starting with a shortest proof.

Case: The proof proceeded with  $\supset L$ .

$$\frac{\neg\neg A \Rightarrow \neg A \quad \neg\neg A, \perp \Rightarrow \neg A}{\neg\neg A \Rightarrow \neg A} \supset L$$

The first premise is identical to the conclusion, so if there were a deduction of that, there would be one without this rule, which is a contradiction to the assumption that we started with the shortest deduction.

□

## 4 Optimizations

We will devote much more time to “optimizations” of the sequent calculus where we try to eliminate redundancies while preserving the same set of theorems. One form of redundancy arises if an antecedent of the premise of the rules are in general not needed to prove the success. For example, in the rule

$$\frac{\Gamma, A \vee B, A \Rightarrow C \quad \Gamma, A \vee B, B \Rightarrow C}{\Gamma, A \vee B \Rightarrow C} \vee L$$

the antecedent  $A \vee B$  is redundant in both premises. In the first premise, for example, we also have  $A$  and  $A$  is stronger than  $A \vee B$  (in the sense that  $A \supset (A \vee B)$ ). In the second premise, it is  $B$  which is stronger than  $A \vee B$ .

In the following summary of the sequent calculus we put [brackets] around the antecedents that could be considered redundant *after optimization*.

$$\begin{array}{c}
 \frac{}{\Gamma, A \Rightarrow A} \text{id} \\
 \frac{\Gamma \Rightarrow A \quad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B} \wedge R \qquad \frac{\Gamma, A \wedge B, A \Rightarrow C}{\Gamma, A \wedge B \Rightarrow C} \wedge L_1 \quad \frac{\Gamma, A \wedge B, B \Rightarrow C}{\Gamma, A \wedge B \Rightarrow C} \wedge L_2 \\
 \\ 
 \frac{}{\Gamma \Rightarrow \top} \top R \qquad \text{no rule } \top L \\
 \\ 
 \frac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \supset B} \supset R \qquad \frac{\Gamma, A \supset B \Rightarrow A \quad \Gamma, [A \supset B], B \Rightarrow C}{\Gamma, A \supset B \Rightarrow C} \supset L \\
 \\ 
 \frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B} \vee R_1 \quad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \vee B} \vee R_2 \qquad \frac{\Gamma, [A \vee B], A \Rightarrow C \quad \Gamma, [A \vee B], B \Rightarrow C}{\Gamma, A \vee B \Rightarrow C} \vee L \\
 \\ 
 \text{no rule } \perp R \qquad \frac{}{\Gamma, \perp \Rightarrow C} \perp L
 \end{array}$$

We could also replace the two left rules for conjunction with

$$\frac{\Gamma, [A \wedge B], A, B \Rightarrow C}{\Gamma, A \wedge B \Rightarrow C} \wedge L$$

## 5 Classical Sequent Calculus

One of Gentzen's remarkable discoveries was the encoding of *classical logic* in the sequent calculus. We already know in natural deduction it can be incorporated by the law of excluded middle, by double negation elimination, or by the rule of indirect proof. All of these are clearly outside the simple beauty of the natural deduction rules as defined by introductions and eliminations.

How do we obtain classical logic? Simply by allowing a sequent to have multiple conclusions! A sequent then has the form  $\Gamma \xrightarrow{\text{CL}} \Delta$ , where  $\Delta$  is also a collection of propositions. Now succedents as well as antecedents in the rules are persistent in all the rules. Remarkably, this is all we need to do!

We can then prove the law of excluded middle as follows, remembering that  $\neg A \triangleq A \supset \perp$ :

$$\frac{\frac{\frac{A \xrightarrow{\text{CL}} A \vee \neg A, A, \neg A}{\xrightarrow{\text{CL}} A \vee \neg A, A, \neg A} \supset R}{\xrightarrow{\text{CL}} A \vee \neg A, A, \neg A} \vee R_2}{\frac{\xrightarrow{\text{CL}} A \vee \neg A, A}{\xrightarrow{\text{CL}} A \vee \neg A}} \vee R_1$$

Somehow, by allowing us to “hedge our bets” about which disjunct is true (first we say “ $A$ ”, send we say “ $\neg A$ ”) and then using the second possibility to establish the first we have circumvented the usual constructive nature of the disjunction.

## References

- [Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.
- [Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

# Lecture Notes on Cut Elimination

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 10  
October 5, 2017

## 1 Introduction

The identity rule of the sequent calculus exhibits one connection between the judgments  $A \text{ left}$  and  $A \text{ right}$ : If we assume  $A \text{ left}$  we can prove  $A \text{ right}$ . In other words, the left rules of the sequent calculus are strong enough so that we can reconstitute a proof of  $A$  from the assumption  $A$ . So the identity theorem (see Section 5) is a global version of the local completeness property for the elimination rules.

The cut theorem of the sequent calculus expresses the opposite: if we have a proof of  $A \text{ right}$  we are licensed to assume  $A \text{ left}$ . This can be interpreted as saying the left rules are not too strong: whatever we can do with the antecedent  $A \text{ left}$  can also be deduced without that, if we know  $A \text{ right}$ . Because  $A \text{ right}$  occurs only as a succedent, and  $A \text{ left}$  only as an antecedent, we must formulate this in a somewhat roundabout manner: If  $\Gamma \Rightarrow A \text{ right}$  and  $\Gamma, A \text{ left} \Rightarrow J$  then  $\Gamma \Rightarrow J$ . In the sequent calculus for pure intuitionistic logic, the only conclusion judgment we are considering is  $C \text{ right}$ , so we specialize the above property.

Because it is very easy to go back and forth between sequent calculus deductions of  $A \text{ right}$  and verifications of  $A \uparrow$ , we can use the cut theorem to show that every true proposition has a verification, which establishes a fundamental, global connection between truth and verifications. While the sequent calculus is a convenient intermediary (and was conceived as such

---

\*With edits by André Platzer

by Gentzen [Gen35]), this theorem can also be established directly using verifications.

## 2 Admissibility of Cut

The cut theorem is one of the most fundamental properties of logic. Because of its central role, we will spend some time on its proof. In lecture we developed the proof and the required induction principle incrementally; here we present the final result as is customary in mathematics. The proof is amenable to formalization in a logical framework; details can be found in a paper by the instructor [Pfe00].

**Theorem 1 (Cut)** *If  $\Gamma \Rightarrow A$  and  $\Gamma, A \Rightarrow C$  then  $\Gamma \Rightarrow C$ .*

**Proof:** By nested inductions on the structure of  $A$ , the derivation  $\mathcal{D}$  of  $\Gamma \Rightarrow A$  and  $\mathcal{E}$  of  $\Gamma, A \Rightarrow C$ . More precisely, we appeal to the induction hypothesis either with a strictly smaller cut formula, or with an identical cut formula and two derivations, one of which is strictly smaller while the other stays the same. The proof is constructive, which means we show how to transform

$$\Gamma \xrightarrow{\mathcal{D}} A \quad \text{and} \quad \Gamma, A \xrightarrow{\mathcal{E}} C \quad \text{to} \quad \Gamma \xrightarrow{\mathcal{F}} C$$

The proof is divided into several classes of cases. More than one case may be applicable, which means that the algorithm for constructing the derivation of  $\Gamma \Rightarrow C$  from the two given derivations is naturally non-deterministic.

**Case:**  $\mathcal{D}$  is an initial sequent,  $\mathcal{E}$  is arbitrary.

$$\mathcal{D} = \frac{}{\Gamma', A \Rightarrow A} \text{id} \quad \text{and} \quad \mathcal{E} = \frac{\Gamma', A, A \Rightarrow C}{\Gamma', A \Rightarrow C}$$

$\Gamma = (\Gamma', A)$	This case
$\Gamma', A, A \Rightarrow C$	Deduction $\mathcal{E}$
$\Gamma', A \Rightarrow C$	By Contraction (see Lecture 9)
$\Gamma \Rightarrow C$	Since $\Gamma = (\Gamma', A)$

**Case:**  $\mathcal{D}$  is arbitrary and  $\mathcal{E}$  is an initial sequent using the cut formula.

$$\Gamma \xrightarrow{\mathcal{D}} A \quad \text{and} \quad \mathcal{E} = \frac{}{\Gamma, A \Rightarrow A} \text{id}$$

$$\begin{array}{c} A = C \\ \Gamma \implies A \end{array}$$

This case  
Deduction  $\mathcal{D}$

**Case:**  $\mathcal{E}$  is an initial sequent *not* using the cut formula.

$$\mathcal{E} = \frac{}{\Gamma', C, A \implies C} \text{id}$$

$$\begin{array}{c} \Gamma = (\Gamma', C) \\ \Gamma', C \implies C \\ \Gamma \implies C \end{array}$$

This case  
By rule id  
Since  $\Gamma = (\Gamma', C)$

In the next set of cases, the cut formula is the principal formula of the final inference in both  $\mathcal{D}$  and  $\mathcal{E}$ . We only show two of these cases.

**Case:**

$$\mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \quad \mathcal{D}_2 \\ \Gamma \implies A_1 \quad \Gamma \implies A_2 \end{array}}{\Gamma \implies A_1 \wedge A_2} \wedge R$$

$$\text{and } \mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \\ \Gamma, A_1 \wedge A_2, A_1 \implies C \end{array}}{\Gamma, A_1 \wedge A_2 \implies C} \wedge L_1$$

$$\begin{array}{c} A = A_1 \wedge A_2 \\ \Gamma, A_1 \implies C \\ \Gamma \implies C \end{array}$$

This case  
By i.h. on  $A_1 \wedge A_2$ ,  $\mathcal{D}$  and  $\mathcal{E}_1$   
By i.h. on  $A_1$ ,  $\mathcal{D}_1$ , and previous line

Actually we have ignored a detail: in the first appeal to the induction hypothesis,  $\mathcal{E}_1$  has an additional hypothesis,  $A_1$ , and therefore does not match the statement of the theorem precisely. However, we can always weaken  $\mathcal{D}$  to include this additional hypothesis without changing the structure of  $\mathcal{D}$  (see the Weakening Theorem in [Lecture 9](#)) and then appeal to the induction hypothesis. We will not be explicit about these trivial weakening steps in the remaining cases.

It is crucial for a well-founded induction that  $\mathcal{E}_1$  is smaller than  $\mathcal{E}$ , so even if the same cut formula and same  $\mathcal{D}$  is used,  $\mathcal{E}_1$  got smaller. Note that we cannot directly appeal to induction hypothesis on  $A_1, \mathcal{D}_1$  and  $\mathcal{E}_1$  because the additional formula  $A_1 \wedge A_2$  might still be used in  $\mathcal{E}_1$ , e.g., by a subsequent use of  $\wedge L_2$ .

**Case:**

$$\mathcal{D} = \frac{\mathcal{D}_2}{\Gamma, A_1 \Rightarrow A_2} \supset R$$

$$\text{and } \mathcal{E} = \frac{\mathcal{E}_1 \quad \mathcal{E}_2}{\Gamma, A_1 \supset A_2 \Rightarrow A_1 \quad \Gamma, A_1 \supset A_2, A_2 \Rightarrow C} \supset L$$

$A = A_1 \supset A_2$	This case
$\Gamma \Rightarrow A_1$	By i.h. on $A_1 \supset A_2$ , $\mathcal{D}$ and $\mathcal{E}_1$
$\Gamma \Rightarrow A_2$	By i.h. on $A_1$ from above and $\mathcal{D}_2$
$\Gamma, A_2 \Rightarrow C$	By i.h. on $A_1 \supset A_2$ , $\mathcal{D}$ and $\mathcal{E}_2$
$\Gamma \Rightarrow C$	By i.h. on $A_2$ from above

Note that the proof constituents of the last step  $\Gamma \Rightarrow C$  may be longer than the original deductions  $\mathcal{D}, \mathcal{E}$ . Hence, it is crucial for a well-founded induction that the cut formula  $A_2$  is smaller than  $A_1 \supset A_2$ .

Finally note the resemblance of these principal cases to the local soundness reductions in harmony arguments for natural deduction.

In the next set of cases, the principal formula in the last inference in  $\mathcal{D}$  is *not* the cut formula. We sometimes call such formulas *side formulas* of the cut.

**Case:** If  $\mathcal{D}$  ended with an  $\wedge L_1$ :

$$\mathcal{D} = \frac{\mathcal{D}_1 \quad \mathcal{E}}{\Gamma', B_1 \wedge B_2, B_1 \Rightarrow A} \wedge L_1 \quad \text{and} \quad \frac{\mathcal{E}}{\Gamma', B_1 \wedge B_2, A \Rightarrow C}$$

$\Gamma = (\Gamma', B_1 \wedge B_2)$	This case
$\Gamma', B_1 \wedge B_2, B_1 \Rightarrow C$	By i.h. on $A$ , $\mathcal{D}_1$ and $\mathcal{E}$
$\Gamma', B_1 \wedge B_2 \Rightarrow C$	By rule $\wedge L_1$
$\Gamma \Rightarrow C$	Since $\Gamma = (\Gamma', B_1 \wedge B_2)$

**Case:**

$$\mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ \Gamma', B_1 \supset B_2 \implies B_1 \quad \Gamma', B_1 \supset B_2, B_2 \implies A \end{array}}{\Gamma', B_1 \supset B_2 \implies A} \supset L$$

$\Gamma = (\Gamma', B_1 \supset B_2)$ $\Gamma', B_1 \supset B_2, B_2 \implies C$ $\Gamma', B_1 \supset B_2 \implies C$ $\Gamma \implies C$	This case By i.h. on $A$ , $\mathcal{D}_2$ and $\mathcal{E}$ By rule $\supset L$ on $\mathcal{D}_1$ and above Since $\Gamma = (\Gamma', B_1 \supset B_2)$
---	--

In the final set of cases,  $A$  is not the principal formula of the last inference in  $\mathcal{E}$ . This overlaps with the previous cases since  $A$  may not be principal on either side. In this case, we appeal to the induction hypothesis on the subderivations of  $\mathcal{E}$  and directly infer the conclusion from the results.

**Case:**

$$\mathcal{D} \quad \text{and} \quad \mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \quad \mathcal{E}_2 \\ \Gamma, A \implies C_1 \quad \Gamma, A \implies C_2 \end{array}}{\Gamma, A \implies C_1 \wedge C_2} \wedge R$$

$C = C_1 \wedge C_2$ $\Gamma \implies C_1$ $\Gamma \implies C_2$ $\Gamma \implies C_1 \wedge C_2$	This case By i.h. on $A$ , $\mathcal{D}$ and $\mathcal{E}_1$ By i.h. on $A$ , $\mathcal{D}$ and $\mathcal{E}_2$ By rule $\wedge R$ on above
--	--

**Case:**

$$\mathcal{D} \quad \text{and} \quad \mathcal{E} = \frac{\mathcal{E}_1}{\Gamma', B_1 \wedge B_2, B_1, A \implies C} \wedge L_1$$

$\Gamma = (\Gamma', B_1 \wedge B_2)$ $\Gamma', B_1 \wedge B_2, B_1 \implies C$ $\Gamma', B_1 \wedge B_2 \implies C$	This case By i.h. on $A$ , $\mathcal{D}$ and $\mathcal{E}_1$ By rule $\wedge L_1$ from above
---	--

□

### 3 Applications of Cut Admissibility

The admissibility of cut, together with the admissibility of identity (see Section 5), complete our program to find global versions of local soundness and completeness. This has many positive consequences. We already have seen that the sequent calculus (without cut!) must be consistent, because there is no sequent proof of  $\perp$ .

If we can translate from arbitrary natural deductions to the sequent calculus, then this also means that natural deduction is consistent, and similarly for other properties such as the disjunction property. Once we have the admissibility of cut, the translation from natural deduction to sequent calculus is surprisingly simple. Note that this is somewhat different from the previous translation that worked on *verifications*: here we are interested in translating arbitrary natural deductions.

**Theorem 2** If  $\frac{\Gamma \quad \begin{array}{c} \mathcal{D} \\ A \text{ true} \end{array}}{A \text{ true}}$  then  $\Gamma \implies A$

**Proof:** By induction on the structure of  $\mathcal{D}$ . For deductions  $\mathcal{D}$  ending in introduction rules, we just replay the corresponding right rule. For example:

$$\text{Case: } \mathcal{D} = \frac{\Gamma \quad \begin{array}{c} \overline{A_1 \text{ true}}^u \\ \mathcal{D}_2 \\ A_2 \text{ true} \end{array}}{A_1 \supset A_2 \text{ true}}$$

$$\begin{array}{ll} \Gamma, A_1 \implies A_2 & \text{By i.h. on } \mathcal{D}_2 \\ \Gamma \implies A_1 \supset A_2 & \text{By rule } \supset R \end{array}$$

For uses of hypotheses, we fill in a use of the identity rule.

$$\text{Case: } \mathcal{D} = \frac{}{A} u$$

$$\Gamma, A \implies A \qquad \qquad \qquad \text{By id}$$

Finally, the tricky cases: elimination rules. In these cases we appeal to the induction hypothesis wherever possible and then use the admissibility of cut!

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \Gamma \\ \mathcal{D}_1 \\ \Gamma \\ \mathcal{D}_2 \\ \hline B \supset A \text{ true} \quad B \text{ true} \end{array}}{A \text{ true}} \supset E$$

$\mathcal{E}_1$  proves  $\Gamma \Rightarrow B \supset A$

By i.h. on  $\mathcal{D}_1$

$\mathcal{E}_2$  proves  $\Gamma \Rightarrow B$

By i.h. on  $\mathcal{D}_2$

To show:  $\Gamma \Rightarrow A$

At this point we realize that the sequent rules “go in the wrong direction”. They are designed to let us prove sequents, rather than take advantage of knowledge, such as  $\Gamma \Rightarrow B \supset A$ .

However, using the admissibility of cut, we can piece together a deduction of  $A$ . First we prove (omitting some redundant antecedents):

$$\mathcal{F} = \frac{\overline{B \Rightarrow B} \text{ id} \quad \overline{A \Rightarrow A} \text{ id}}{B \supset A, B \Rightarrow A} \supset L$$

Then (leaving some trivial instances of weakening implicit):

$\mathcal{F}_1$  proves  $\Gamma, B \Rightarrow A$   
 $\Gamma \Rightarrow A$

By adm. of cut on  $\mathcal{E}_1$  and  $\mathcal{F}$   
 By adm. of cut on  $\mathcal{E}_2$  and  $\mathcal{F}_1$

where the last line is what we needed to show.

□

The translation from sequent proofs to verifications is quite straightforward, so we omit it here. But chaining these proof translations together we find that every true propositions  $A$  (as defined by natural deduction) has a verification. This closes the loop on our understanding of the connections between natural deductions, sequent proofs, and verifications.

## 4 Cut Elimination<sup>1</sup>

Gentzen’s original presentation of the sequent calculus included an inference rule for cut. We write  $\Gamma \xrightarrow{\text{cut}} A$  for this system, which is just like

---

<sup>1</sup>This material not covered in lecture

$\Gamma \Rightarrow A$ , with the additional rule

$$\frac{\Gamma \xrightarrow{\text{cut}} A \quad \Gamma, A \xrightarrow{\text{cut}} C}{\Gamma \xrightarrow{\text{cut}} C} \text{cut}$$

The advantage of this calculus is that it more directly corresponds to natural deduction in its full generality, rather than verifications, because just like in natural deduction, the cut rule makes it possible to prove an arbitrary other  $A$  from the available assumptions  $\Gamma$  (left premise) and then use that  $A$  as an additional assumption in the rest of the proof (right premise). The disadvantage is that it cannot easily be seen as capturing the meaning of the connectives by inference rules, because with the rule of cut the meaning of  $C$  might depend on the meaning of any other proposition  $A$  (possibly even including  $C$  as a subformula).

In order to clearly distinguish between the two kinds of calculi, the one we presented is sometimes called the *cut-free sequent calculus*, while Gentzen's calculus would be a *sequent calculus with cut*. The theorem connecting the two is called *cut elimination*: for any deduction in the sequent calculus with cut, there exists a cut-free deduction of the same sequent. The proof is a straightforward induction on the structure of the deduction, appealing to the cut theorem in one crucial place.

**Theorem 3 (Cut Elimination)** *If  $\mathcal{D}$  is a deduction of  $\Gamma \xrightarrow{\text{cut}} C$  possibly using the cut rule, then there exists a cut-free deduction  $\mathcal{D}'$  of  $\Gamma \xrightarrow{\text{cut}} C$ .*

**Proof:** By induction on the structure of  $\mathcal{D}$ . In each case, we appeal to the induction hypothesis on all premises and then apply the same rule to the result. The only interesting case is when a cut rule is encountered.

**Case:**

$$\mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \qquad \mathcal{D}_2 \\ \Gamma \xrightarrow{\text{cut}} A \quad \Gamma, A \xrightarrow{\text{cut}} C \end{array}}{\Gamma \xrightarrow{\text{cut}} C} \text{cut}$$

$$\begin{array}{ll} \Gamma \Rightarrow A & \text{without cut} \\ \Gamma, A \Rightarrow C & \text{without cut} \\ \Gamma \Rightarrow C & \end{array}$$

By i.h. on  $\mathcal{D}_1$   
By i.h. on  $\mathcal{D}_2$   
By the Cut Theorem

□

## 5 Identity<sup>2</sup>

We permit the identity rule for all propositions. However, the version of this rule for just atomic propositions  $P$  is strong enough. We write  $\Gamma \xrightarrow{\text{id}} A$  for this restricted system. In this restricted system, the rule for arbitrary propositions  $A$  is *admissible*, that is, each instance of the rule can be deduced. We call this the *identity theorem* because it shows that from an assumption  $A$  we can prove the identical conclusion  $A$ .

**Theorem 4 (Identity)** *For any proposition  $A$ , we have  $A \xrightarrow{\text{id}} A$ .*

**Proof:** By induction on the structure of  $A$ . We show several representative cases and leave the remaining ones to the reader.

Case:  $A = P$  for an atomic proposition  $P$ . Then

$$\frac{}{P \xrightarrow{\text{id}} P} \text{id}$$

Case:  $A = A_1 \wedge A_2$ . Then

By i.h. on  $A_1$  and weakening    By i.h. on  $A_2$  and weakening

$$\frac{\begin{array}{c} A_1 \wedge A_2, A_1 \xrightarrow{\text{id}} A_1 \\ \hline A_1 \wedge A_2 \xrightarrow{\text{id}} A_1 \end{array} \wedge L_1 \quad \begin{array}{c} A_1 \wedge A_2, A_2 \xrightarrow{\text{id}} A_2 \\ \hline A_1 \wedge A_2 \xrightarrow{\text{id}} A_2 \end{array} \wedge L_2}{A_1 \wedge A_2 \xrightarrow{\text{id}} A_1 \wedge A_2} \wedge R$$

Case:  $A = A_1 \supset A_2$ . Then

By i.h. on  $A_1$  and weakening    By i.h. on  $A_2$  and weakening

$$\frac{\begin{array}{c} A_1 \supset A_2, A_1 \xrightarrow{\text{id}} A_1 \\ \hline A_1 \supset A_2, A_1, A_2 \xrightarrow{\text{id}} A_2 \end{array} \supset L \quad \begin{array}{c} A_1 \supset A_2, A_1 \xrightarrow{\text{id}} A_2 \\ \hline A_1 \supset A_2 \xrightarrow{\text{id}} A_1 \supset A_2 \end{array} \supset R}{A_1 \supset A_2 \xrightarrow{\text{id}} A_1 \supset A_2}$$

Case:  $A = \perp$ . Then

$$\frac{}{\perp \xrightarrow{\text{id}} \perp} \perp L$$

---

<sup>2</sup>this section not covered in lecture

□

The identity theorem is the global version of the local completeness property for each individual connective. Local completeness shows that a connective can be re-verified from a proof that gives us license to use it, which directly corresponds to  $A \xrightarrow{\text{id}} A$ . One can recognize the local expansion as embodied in each case of the inductive proof of identity.

## References

- [Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.
- [Pfe00] Frank Pfenning. Structural cut elimination I. Intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

# Lecture Notes on Propositional Theorem Proving

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 11  
October 10, 2017

## 1 Introduction

The sequent calculus we have introduced so far maintains a close correspondence to natural deductions or, more specifically, to verifications. One consequence is *persistence of antecedents*: once an assumption has been introduced in the course of a deduction, it will remain available in any sequent above this point. While this is appropriate in a foundational calculus, it is not ideal for proof search since rules can be applied over and over again without necessarily making progress. We therefore develop a second sequent calculus and then a third in order to make the process of bottom-up search for a proof more efficient by reducing unnecessary choices in proof search. By way of the previous link of the sequent calculus with verification-style natural deductions, this lecture will, thus, give rise to a more efficient way of coming up with natural deduction proofs.

This lecture marks the begin of a departure from the course of the lectures so far, which, broadly construed, focused on understanding what a constructive proof is and what can be read off or done once one has such a proof. Now we begin to move toward the question of how to find such a proof in the first place.

More ambitiously, we are looking for a *decision procedure* for intuitionistic propositional logic. Specifically, we would like to prove that for every proposition  $A$ , either  $\Rightarrow A$  or not  $\Rightarrow A$ . Based on experience, we suspect this could be proved by induction on  $A$ , but this will fail for various

---

\*With edits by André Platzer

reasons. We somehow need to generalize it to prove that for *every sequent*, either  $\Gamma \implies A$  or not. That, however, has its own problems because the premises of the rules are more complex than the conclusion so it is not clear how one might apply an induction hypothesis.

First order of business then is to find a new, more restrictive system that eliminates redundancy and makes the premises of the rules smaller than the conclusion. This restricted sequent calculus will not quite satisfy our goal yet, but be useful nonetheless.

The second step will be to refine our analysis of the rules to see if we can design a calculus were *all* premises are smaller than the conclusion in some well-founded ordering. Dyckhoff [Dyc92] noticed that we can make progress by considering the possible forms of the antecedent of the implication. In each case we can write a special-purpose rule for which the premises are smaller than conclusion. The result is a beautiful calculus which Dyckhoff calls *contraction-free* because there is no rule of contraction, and, furthermore, the principal formula of each left rule is consumed as part of the rule application rather than copied to any premise, so we never duplicate reasoning (which we could if there were a contraction rule).

## 2 A More Restrictive Sequent Calculus<sup>1</sup>

Ideally, once we have applied an inference rule during proof search (that is, bottom-up), we should not have to apply the same rule again to the same proposition. Since all rules decompose formulas, if we had such a sequent calculus, we would have a simple and clean decision procedure. As it turns out, there is a fly in the ointment, but let us try to derive such a system.

We write  $\Gamma \rightarrow C$  for a sequent whose deductions try to eliminate principal formulas as much as possible. We keep the names of the rules, since they are largely parallel to the rules of the original sequent calculus,  $\Gamma \implies C$ .

**Conjunction.** The right rule works as before; the left rule extracts *both* conjuncts so that the conjunction itself is no longer needed.

$$\frac{\Gamma \rightarrow A \quad \Gamma \rightarrow B}{\Gamma \rightarrow A \wedge B} \wedge R \qquad \frac{\Gamma, A, B \rightarrow C}{\Gamma, A \wedge B \rightarrow C} \wedge L$$

---

<sup>1</sup>This calculus was mentioned in an earlier lecture, without proof. We show it here as a starting point for the contraction-free calculus, as we did in lecture.

Observe that for both rules, all premises have smaller sequents than the conclusion if one counts the number of connectives in a sequent. So applying either rule obviously made progress toward simplifying the sequent.

It is easy to see that these rules are sound with respect to the ordinary sequent calculus rules. Soundness here is the property that if  $\Gamma \rightarrow C$  then  $\Gamma \Rightarrow C$ . This is straightforward since  $\wedge R$  is the same rule and  $\wedge L$  is the same as  $\wedge L_1$  followed by  $\wedge L_2$  followed by weakening the original  $A \wedge B$  away. Completeness is generally more difficult. What we want to show is that if  $\Gamma \Rightarrow C$  then also  $\Gamma \rightarrow C$ , where the rules for the latter sequents are more restrictive, by design. The proof of this will eventually proceed by induction on the structure of the given deduction  $\mathcal{D}$  and appeal to lemmas on the restrictive sequent calculus. For example:

**Case: (of completeness proof)**

$$\mathcal{D} = \frac{\mathcal{D}_1}{\Gamma, A \wedge B \Rightarrow C} \wedge L_1$$

$\Gamma, A \wedge B, A \rightarrow C$ $\Gamma, A, B \rightarrow A$ $\Gamma, A \wedge B \rightarrow A$ $\Gamma, A \wedge B \rightarrow C$	By i.h. on $\mathcal{D}_1$ By identity for $\rightarrow$ By $\wedge L$ By cut for $\rightarrow$
---	--

The induction hypothesis is applicable to  $\mathcal{D}_1$  because, even if it is a longer sequent,  $\mathcal{D}_1$  is a shorter proof than  $\mathcal{D}$ . We see that identity and cut for the restricted sequent calculus are needed to show completeness in the sense described above. Fortunately, they hold (see further notes at the end of this section). We will not formally justify many of the rules, but give informal justifications or counterexamples.

**Truth.** There is a small surprise here, in that, unlike in natural deduction which had no elimination rule for  $\top$ , we can have a left rule for  $\top$ , which eliminates it from the antecedents to make progress (cleanup). It is analogous to the zero-ary case of conjunction.

$$\frac{}{\Gamma \rightarrow \top} \top R \quad \frac{\Gamma \rightarrow C}{\Gamma, \top \rightarrow C} \top L$$

**Atomic propositions.** They are straightforward, since the initial sequents do not change.

$$\frac{}{\Gamma, P \rightarrow P} \text{id}$$

**Disjunction.** The right rules do not change; in the left rule we can eliminate the principal formula.

$$\frac{\Gamma \rightarrow A}{\Gamma \rightarrow A \vee B} \vee R_1 \quad \frac{\Gamma \rightarrow B}{\Gamma \rightarrow A \vee B} \vee R_2 \quad \frac{\Gamma, A \rightarrow C \quad \Gamma, B \rightarrow C}{\Gamma, A \vee B \rightarrow C} \vee L$$

Intuitively, the assumption  $A \vee B$  can be eliminated from both premises of the  $\vee L$  rule, because the new assumptions  $A$  and  $B$  are stronger. More formally:

**Case: (of completeness proof)**

$$\mathcal{D} = \frac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Gamma, A \vee B, A \rightarrow C \quad \Gamma, A \vee B, B \rightarrow C} \vee L$$

$$\begin{array}{ll}
 \Gamma, A \vee B, A \rightarrow C & \text{By i.h. on } \mathcal{D}_1 \\
 \Gamma, A \rightarrow A & \text{By identity for } \rightarrow \\
 \Gamma, A \rightarrow A \vee B & \text{By } \vee R_1 \\
 \Gamma, A \rightarrow C & \text{By cut for } \rightarrow \\
 \Gamma, A \vee B, B \rightarrow C & \text{By i.h. on } \mathcal{D}_2 \\
 \Gamma, B \rightarrow B & \text{By identity for } \rightarrow \\
 \Gamma, B \rightarrow A \vee B & \text{By } \vee R_2 \\
 \Gamma, B \rightarrow C & \text{By cut for } \rightarrow \\
 \Gamma, A \vee B \rightarrow C & \text{By rule } \vee L
 \end{array}$$

**Falsehood.** There is no right rule, and the left rule has no premise, which means it transfers directly.

$$\text{no } \perp R \text{ rule} \quad \frac{}{\Gamma, \perp \rightarrow C} \perp L$$

**Implication.** In all the rules so far, all premises have fewer connectives than the conclusion. For implication, we will not be able to maintain this property.

$$\frac{\Gamma, A \rightarrow B}{\Gamma \rightarrow A \supset B} \supset R \quad \frac{\Gamma, A \supset B \rightarrow A \quad \Gamma, B \rightarrow C}{\Gamma, A \supset B \rightarrow C} \supset L$$

Here, the assumption  $A \supset B$  persists in the first premise but not in the second. While the assumption  $B$  is more informative than  $A \supset B$ , so only  $B$  is kept in the second premise, this is not the case in the first premise. Unfortunately,  $A \supset B$  may be needed again in that branch of the proof. An example which requires the implication more than once is  $\rightarrow \neg\neg(A \vee \neg A)$ , where  $\neg A = A \supset \perp$  as usual. Without that additional assumption (marked in red below), the proof would not work:

$$\begin{array}{c} \frac{\neg(A \vee \neg A), A \rightarrow A}{\neg(A \vee \neg A), A \rightarrow A \vee \neg A} \text{id} \\ \frac{\neg(A \vee \neg A), A \rightarrow A \vee \neg A \quad \frac{A, \perp \rightarrow \perp}{\perp \rightarrow \perp} \perp L}{\neg(A \vee \neg A), A \rightarrow \perp} \supset L \\ \frac{\neg(A \vee \neg A), A \rightarrow \perp}{\neg(A \vee \neg A) \rightarrow \neg A} \supset R \\ \frac{\neg(A \vee \neg A) \rightarrow \neg A}{\neg(A \vee \neg A) \rightarrow A \vee \neg A} \vee R_2 \quad \frac{}{\perp \rightarrow \perp} \perp L \\ \frac{\neg(A \vee \neg A) \rightarrow A \vee \neg A \quad \frac{\perp \rightarrow \perp}{\perp \rightarrow \perp} \perp L}{\neg(A \vee \neg A) \rightarrow \perp} \supset L \\ \frac{\neg(A \vee \neg A) \rightarrow \perp}{\rightarrow \neg\neg(A \vee \neg A)} \supset R \end{array}$$

Now all rules have smaller premises (if one counts the number of logical constants and connectives in them) except for the  $\supset L$  rule. We will address the issue with  $\supset L$  in Section 4.

Nevertheless, we can interpret the rules as a decision procedure if we make the observation that in bottom-up proof search we are licensed to fail a branch if along it we have a repeating sequent. If there were a deduction, we would be able to find it applying a different choice at an earlier sequent, lower down in the incomplete deduction. If there is a proof with repeating sequents, there also is a proof without repeating sequents, by applying the proof that was used for the later occurrence of the repeating sequent already to the first occurrence of said sequent. If we also apply contraction (which is admissible in the restricted sequent calculus) to argue that we can remove duplicate formulas from the antecedent, then there are only finitely many sequents because antecedents and succedents are composed only of subformulas of our original proof goal.

One can be much more efficient in loop checking than this [How98, Chapter 4], but just to see that intuitionistic propositional calculus is decidable, this is sufficient. In fact, we could have made this observation on the original sequent calculus, although it would be even further from a realistic implementation.

### 3 Metatheory of the Restricted Sequent Calculus

We only enumerate the basic properties.

**Theorem 1 (Weakening)** *If  $\Gamma \rightarrow C$  then  $\Gamma, A \rightarrow C$  with a structurally identical deduction.*

**Theorem 2 (Atomic contraction)** *If  $\Gamma, P, P \rightarrow C$  then  $\Gamma, P \rightarrow C$  with a structurally identical deduction*

**Theorem 3 (Identity)**  *$A \rightarrow A$  for any proposition  $A$ .*

**Proof:** By induction on the structure of  $A$ . □

**Theorem 4 (Cut)** *If  $\Gamma \rightarrow A$  and  $\Gamma, A \rightarrow C$  then  $\Gamma \rightarrow C$*

**Proof:** Analogous to the proof for the ordinary sequent calculus in [Lecture 8](#). In the case where the first deduction is initial, we use atomic contraction. □

**Theorem 5 (Contraction)** *If  $\Gamma, A, A \rightarrow C$  then  $\Gamma, A \rightarrow C$ .*

**Proof:**  $\Gamma, A \rightarrow A$  by identity and weakening. Therefore  $\Gamma, A \rightarrow C$  by cut. □

**Theorem 6 (Soundness wrt.  $\implies$ )** *If  $\Gamma \rightarrow A$  then  $\Gamma \implies A$ .*

**Proof:** By induction on the structure of the given deduction. □

**Theorem 7 (Completeness wrt.  $\implies$ )** *If  $\Gamma \implies A$  then  $\Gamma \rightarrow A$ .*

**Proof:** By induction on the structure of the given deduction, appealing to identity and cut in many cases. See the cases for  $\wedge L_1$  and  $\vee L$  in the previous section. □

We repeat the rules of the restrictive sequent calculus here for reference.

$$\begin{array}{c}
 \overline{\Gamma, P \longrightarrow P} \text{ id} \\
 \frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B} \wedge R \quad \frac{\Gamma, A, B \longrightarrow C}{\Gamma, A \wedge B \longrightarrow C} \wedge L \\
 \frac{}{\Gamma \longrightarrow \top} \top R \quad \frac{\Gamma \longrightarrow C}{\Gamma, \top \longrightarrow C} \top L \\
 \frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B} \vee R_1 \quad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B} \vee R_2 \quad \frac{\Gamma, A \longrightarrow C \quad \Gamma, B \longrightarrow C}{\Gamma, A \vee B \longrightarrow C} \vee L \\
 \text{no } \perp R \text{ rule} \quad \frac{}{\Gamma, \perp \longrightarrow C} \perp L \\
 \frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B} \supset R \quad \frac{\Gamma, A \supset B \longrightarrow A \quad \Gamma, B \longrightarrow C}{\Gamma, A \supset B \longrightarrow C} \supset L
 \end{array}$$

## 4 Refining the Left Rule for Implication

In order to find a more efficient form of the problematic rule  $\supset L$ , we consider each possibility for the antecedent of the implication in turn. We will start with more obvious cases to find out the principles behind the design of the rules.

**Truth.** Consider a sequent

$$\Gamma, \top \supset B \longrightarrow C$$

Can we find a simpler proposition expressing the same as  $\top \supset B$ ? Yes, namely just  $B$ , since  $(\top \supset B) \equiv B$ . So we can propose the following specialized rule:

$$\frac{\Gamma, B \longrightarrow C}{\Gamma, \top \supset B \longrightarrow C} \top \supset L$$

This rule derives from  $\supset L$  and  $\top R$ , which are both sound.

**Falsehood.** Consider a sequent

$$\Gamma, \perp \supset B \longrightarrow C.$$

Can we find a simpler proposition expressing the same contents? Yes, namely  $\top$ , since  $(\perp \supset B) \equiv \top$ . But  $\top$  on the left-hand side can be eliminated by  $\top L$ , so we can specialize the general rule as follows:

$$\frac{\Gamma \longrightarrow C}{\Gamma, \perp \supset B \longrightarrow C} \perp \supset L$$

Soundness of this rule also follows from weakening. Are we losing information compared to applying  $\supset L$  here? No because that would require a proof of  $\Gamma, \perp \supset B \longrightarrow \perp$  which will succeed if  $\perp$  can be proved from  $\Gamma$ , but then there also is a direct proof without using  $\perp \supset B$ .

**Disjunction.** Now we consider a sequent

$$\Gamma, (A_1 \vee A_2) \supset B \longrightarrow C$$

Again, we have to ask if there is a simpler equivalent formula we can use instead of  $(A_1 \vee A_2) \supset B$ . If we consider the  $\vee L$  rule, we might consider  $(A_1 \supset B) \wedge (A_2 \supset B)$ . A little side calculation confirms that, indeed,

$$((A_1 \vee A_2) \supset B) \equiv ((A_1 \supset B) \wedge (A_2 \supset B))$$

The computational intuition is that getting a  $B$  out of having either a  $A_1$  or an  $A_2$  is equivalent to separate ways of getting a  $B$  out of a  $A_1$  as well as a way of getting a  $B$  out of an  $A_2$ . We can exploit this, playing through the rules as follows

$$\frac{\begin{array}{c} \Gamma, A_1 \supset B, A_2 \supset B \longrightarrow C \\ \Gamma, (A_1 \supset B) \wedge (A_2 \supset B) \longrightarrow C \\ \hline \Gamma, (A_1 \vee A_2) \supset B \longrightarrow C \end{array}}{\Gamma, (A_1 \vee A_2) \supset B \longrightarrow C} \wedge L \text{ equiv}$$

This suggests the specialized rule

$$\frac{\Gamma, A_1 \supset B, A_2 \supset B \longrightarrow C}{\Gamma, (A_1 \vee A_2) \supset B \longrightarrow C} \vee \supset L$$

The question is whether the premise is really smaller than the conclusion in some well-founded measure. We note that both  $A_1 \supset B$  and  $A_2 \supset B$  are

smaller than the original formula  $(A_1 \vee A_2) \supset B$ . Replacing one element in a multiset by several, each of which is strictly smaller according to some well-founded ordering, induces another well-founded ordering on multisets [DM79]. So, the premises are indeed smaller in the multiset ordering. Operationally, the effect of  $\vee\supset L$  is to separately consider the smaller implications  $A_1 \supset B$  and  $A_2 \supset B$ .

**Conjunction.** Next we consider

$$\Gamma, (A_1 \wedge A_2) \supset B \longrightarrow C$$

In this case we can create an equivalent formula by currying using that  $(A_1 \wedge A_2) \supset B \equiv A_1 \supset (A_2 \supset B)$ .

$$\frac{\Gamma, A_1 \supset (A_2 \supset B) \longrightarrow C}{\Gamma, (A_1 \wedge A_2) \supset B \longrightarrow C} \wedge\supset L$$

This formula is not strictly smaller, but we can make it so by giving conjunction a weight of 2 while counting implications as 1. Fortunately, this weighting does not conflict with any of the other rules we have. Operationally, the effect of  $\wedge\supset L$  is to first consider what to make of the first assumed conjunct  $A_1$  by the other rules and then subsequently consider the second conjunct  $A_2$ .

**Atomic propositions.** How do we use an assumption  $P \supset B$ ? We can conclude if we also know  $P$ , so we restrict the rule to the case where  $P$  is already among the assumption.

$$\frac{P \in \Gamma \quad \Gamma, B \longrightarrow C}{\Gamma, P \supset B \longrightarrow C} P\supset L$$

Clearly, the premise is smaller than the conclusion. If we were to use  $\supset L$  instead,  $P \supset B$  would remain in the first premise. The intuitive reason why we do not have to keep it is because the only way to make use of  $P \supset B$  is to produce a  $P$ . But if we have such an atomic  $P$ , the above rule already establishes  $B$ . Note that, unlike a premise  $\Gamma \longrightarrow P$ , the premise  $P \in \Gamma$  will obviously never search for possible proof rule applications within  $\Gamma$ . Indeed, those would not be useful, because we might as well apply them first before splitting into two premises.

**Implication.** Last, but not least, we consider the case

$$\Gamma, (A_1 \supset A_2) \supset B \longrightarrow C$$

We start by playing through the left rule  $\supset L$  for this particular case because, as we have already seen, an implication on the left does not directly simplify when interacting with another implication.

$$\frac{\frac{\Gamma, (A_1 \supset A_2) \supset B, A_1 \longrightarrow A_2}{\Gamma, (A_1 \supset A_2) \supset B \longrightarrow A_1 \supset A_2} \supset R \quad \Gamma, B \longrightarrow C}{\Gamma, (A_1 \supset A_2) \supset B \longrightarrow C} \supset L$$

The second premise is smaller and does not require any further attention. For the first premise, we need to find a smaller formula that is equivalent to  $((A_1 \supset A_2) \supset B) \wedge A_1$ . The conjunction here is a representation of two distinguished formulas in the context. Fortunately, we find

$$((A_1 \supset A_2) \supset B) \wedge A_1 \equiv (A_2 \supset B) \wedge A_1$$

which can be checked easily since  $A_1 \supset A_2$  is equivalent to  $A_2$  if we already have  $A_1$ . This leads to the specialized rule

$$\frac{\Gamma, A_2 \supset B, A_1 \longrightarrow A_2 \quad \Gamma, B \longrightarrow C}{\Gamma, (A_1 \supset A_2) \supset B \longrightarrow C} \supset \supset L$$

Indeed, all premises of  $\supset \supset L$  are simpler now, because  $A_2 \supset B$  has strictly less operators than  $(A_1 \supset A_2) \supset B$  and its operators are of the same weight.

There is a minor variation of this rule, which is also both sound and complete, and the premises are all smaller (by the multiset ordering) than the conclusion.

$$\frac{\Gamma, A_2 \supset B \longrightarrow A_1 \supset A_2 \quad \Gamma, B \longrightarrow C}{\Gamma, (A_1 \supset A_2) \supset B \longrightarrow C} \supset \supset L$$

They are equivalent because, in general,  $\Gamma \longrightarrow A_1 \supset A_2$  iff  $\Gamma, A_1 \longrightarrow A_2$ .

This concludes the presentation of the specialized rules so that the only rule that kept its principal formula around,  $\supset L$ , is no longer needed since all forms of implications are covered. The complete set of rule is summarized in Figure 1.

Even though these rules can be interpreted as defining a decision procedure, such a procedure would not be practical except for small examples because there is too much nondeterminism in choosing which rule to apply when. We will discuss this in the next lecture.

$$\begin{array}{c}
 \frac{P \in \Gamma}{\Gamma \rightarrow P} \text{id} \\
 \\
 \frac{\Gamma \rightarrow A \quad \Gamma \rightarrow B}{\Gamma \rightarrow A \wedge B} \wedge R \quad \frac{\Gamma, A, B \rightarrow C}{\Gamma, A \wedge B \rightarrow C} \wedge L \\
 \\
 \frac{}{\Gamma \rightarrow \top} \top R \quad \frac{\Gamma \rightarrow C}{\Gamma, \top \rightarrow C} \top L \\
 \\
 \frac{\Gamma \rightarrow A}{\Gamma \rightarrow A \vee B} \vee R_1 \quad \frac{\Gamma \rightarrow B}{\Gamma \rightarrow A \vee B} \vee R_2 \quad \frac{\Gamma, A \rightarrow C \quad \Gamma, B \rightarrow C}{\Gamma, A \vee B \rightarrow C} \vee L \\
 \\
 \text{no } \perp R \text{ rule} \quad \frac{}{\Gamma, \perp \rightarrow C} \perp L \\
 \\
 \frac{\Gamma, A \rightarrow B}{\Gamma \rightarrow A \supset B} \supset R \\
 \\
 \frac{P \in \Gamma \quad \Gamma, B \rightarrow C}{\Gamma, P \supset B \rightarrow C} \supset L \\
 \\
 \frac{\Gamma, A_1 \supset (A_2 \supset B) \rightarrow C}{\Gamma, (A_1 \wedge A_2) \supset B \rightarrow C} \wedge \supset L \quad \frac{\Gamma, B \rightarrow C}{\Gamma, \top \supset B \rightarrow C} \top \supset L \\
 \\
 \frac{\Gamma, A_1 \supset B, A_2 \supset B \rightarrow}{\Gamma, (A_1 \vee A_2) \supset B \rightarrow C} \vee \supset L \quad \frac{\Gamma \rightarrow C}{\Gamma, \perp \supset B \rightarrow C} \perp \supset L \\
 \\
 \frac{\Gamma, A_2 \supset B, A_1 \rightarrow A_2 \quad \Gamma, B \rightarrow C}{\Gamma, (A_1 \supset A_2) \supset B \rightarrow C} \supset \supset L
 \end{array}$$

Figure 1: Contraction-free sequent calculus

## References

- [DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.
- [Dyc92] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:795–807, 1992.
- [How98] Jacob M. Howe. *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St. Andrews, Scotland, 1998.

# Lecture Notes on Inversion

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 12  
October 12, 2017

## 1 Introduction

The contraction-free sequent calculus can be seen as describing a decision procedure for intuitionistic propositional logic. Great! But as soon as we have a sequent with many antecedents, there are many choices of rules to apply. Unless we somehow “optimize” we would have to try them all for each sequent we are trying to prove. This turns out not to be feasible except for very small examples.

Fortunately, some rules have the property that we can *always* apply them without having to consider alternatives. Loosely speaking, this is because whenever the conclusion is provable, so are all the premises. We call such rules *invertible* and the proof search strategy that first applies all such rules *inversion*. In this lecture we develop a calculus in which inversion is “built-in” in the sense that the only legal deductions are those that do apply inversions eagerly.

## 2 Invertible Rules

The restrictive sequent calculus in the previous section is a big improvement, but if we use it directly to implement a search procedure it is hopelessly inefficient. The problem is that for any goal sequent, any left or right rule might be applicable. But the application of a rule changes the sequent

---

\*With edits by André Platzer

just a little—most formulas are preserved and we are faced with the same choices at the next step. Eliminating this kind of inefficiency is crucial for a practical theorem proving procedure.

The first observation, to be refined later, is that certain rules are *invertible*, that is, the premises hold iff the conclusion holds. This is powerful, because we can apply the rule and never look back and consider any other choice.

As an example of an invertible rule, consider  $\wedge R$  again:

$$\frac{\Gamma \rightarrow A \quad \Gamma \rightarrow B}{\Gamma \rightarrow A \wedge B} \wedge R$$

The premises already imply the conclusion since the rule is sound. So for  $\wedge R$  to be invertible means that if the conclusion holds then both premises hold as well. That is, we have to show: *If  $\Gamma \rightarrow A \wedge B$  then  $\Gamma \rightarrow A$  and  $\Gamma \rightarrow B$* , which is the opposite of what the rule itself expresses. Fortunately, this follows easily by cut, since  $\Gamma, A \wedge B \rightarrow A$  and  $\Gamma, A \wedge B \rightarrow B$ .

$$\frac{\Gamma \rightarrow A \wedge B \quad \frac{\overline{\Gamma, A, B \rightarrow A} \text{ id}}{\Gamma, A \wedge B \rightarrow A} \wedge L}{\dots \dots \dots \text{ cut}} \Gamma \rightarrow A$$

In order to formalize the strategy of applying inversions eagerly, without backtracking over the choices of which invertible rules to try, we refine the restricted sequent calculus further into two, mutually dependent forms of sequents.

$$\begin{array}{ll} \Gamma^- ; \Omega \xrightarrow{R} C & \text{Decompose } C \text{ on the right} \\ \Gamma^- ; \Omega \xrightarrow{L} C^+ & \text{Decompose } \Omega \text{ on the left} \end{array}$$

Here,  $\Omega$  is an *ordered context* (say, a stack) that we only access at the right end. To make this stand out in the notation, we write  $\Omega \cdot A$  instead of  $\Omega, A$  when building up or decomposing ordered contexts.

$\Gamma^-$  is a context restricted to those formulas whose left rules are *not* invertible, and  $C^+$  is a formula whose right rule is *not* invertible. Both types of sequents can also contain atoms.

Only left decompositions  $\Gamma^- ; \Omega \xrightarrow{L} C^+$  are restricted to have a formula with a connective of a non-invertible right-rule. Right decompositions  $\Gamma^- ; \Omega \xrightarrow{R} C$  are unrestricted. The idea is that decompositions in the

ordered context  $\Omega$  should be preferred when the succedent is of the non-invertible form  $C^+$  so does not have a canonical search-free decomposition. Overall, actions in the ordered context  $\Omega$  will turn out to be deterministic while those for  $\Gamma^-$  involve decisions and search. That gives eager invertible decompositions and lazy search for non-invertibles.

After we have developed the rules we will summarize the forms of  $\Gamma^-$  and  $C^+$ . We refer to this as the *inversion calculus*. Rather than organizing the presentation by connective, we will follow the judgments, starting on the right. That presentation order will enable us to emphasize the intended search order and exhaustiveness of the resulting procedure.

**Right inversion.** We decompose conjunction, truth, and implication eagerly on the right and on the left, because both rules are invertible and can easily be checked.

$$\frac{\Gamma^- ; \Omega \xrightarrow{R} A \quad \Gamma^- ; \Omega \xrightarrow{R} B}{\Gamma^- ; \Omega \xrightarrow{R} A \wedge B} \wedge R \quad \frac{}{\Gamma^- ; \Omega \xrightarrow{R} \top} \top R \quad \frac{\Gamma^- ; \Omega \cdot A \xrightarrow{R} B}{\Gamma^- ; \Omega \xrightarrow{R} A \supset B} \supset R$$

If we encounter an atomic formula, we switch to inverting on the left, since the identity rule is not invertible. Once could optimize further by checking if  $P$  is in the context and only start left inversion if it is not.

$$\frac{\Gamma^- ; \Omega \xrightarrow{L} P}{\Gamma^- ; \Omega \xrightarrow{R} P} LR_P$$

If we encounter disjunction or falsehood, we punt and switch to left inversion.

$$\frac{\Gamma^- ; \Omega \xrightarrow{L} A \vee B}{\Gamma^- ; \Omega \xrightarrow{R} A \vee B} LR_V \quad \frac{\Gamma^- ; \Omega \xrightarrow{L} \perp}{\Gamma^- ; \Omega \xrightarrow{R} \perp} LR_{\perp}$$

Disjunctions would need a commitment whether their left or their right disjunct is proved. Switching to right decomposition postpones that choice until we maximize what we know. Note how the right inversion rules really only switch to left decomposition for non-invertible succedents  $C^+$ . Also suddenly there is a rule for  $\perp$  on the right, but it merely switches mode to left inversion, so no need to panic.

**Left inversion.** The next phase performs left inversion at the right end of the ordered context  $\Omega$ . Note that for each logical connective or constant, there is exactly one rule to apply.

$$\begin{array}{c} \frac{\Gamma^- ; \Omega \cdot A \cdot B \xrightarrow{L} C^+}{\Gamma^- ; \Omega \cdot (A \wedge B) \xrightarrow{L} C^+} \wedge L \quad \frac{\Gamma^- ; \Omega \xrightarrow{L} C^+}{\Gamma^- ; \Omega \cdot \top \xrightarrow{L} C^+} \top L \\[10pt] \frac{\Gamma^- ; \Omega \cdot A \xrightarrow{L} C^+ \quad \Gamma^- ; \Omega \cdot B \xrightarrow{L} C^+}{\Gamma^- ; \Omega \cdot (A \vee B) \xrightarrow{L} C^+} \vee L \quad \frac{}{\Gamma^- ; \Omega \cdot \perp \xrightarrow{L} C^+} \perp L \end{array}$$

Observe how helpful it is that the succedent of  $\vee L$  is already decomposed to  $C^+$  so has no invertible right rule, otherwise we would have to repeat the same effort decomposing the succedent by right inversion on both premises of  $\vee L$ . For atomic formulas, we just move them into the noninvertible context, since the identity is not invertible. We could optimize further by looking of  $P$  was equal to  $C^+$  succeed if so.

$$\frac{\Gamma^- , P ; \Omega \xrightarrow{L} C^+}{\Gamma^- ; \Omega \cdot P \xrightarrow{L} C^+} \text{shift}_P$$

Finally, in the inversion phase, if the formula on the left is an implication, which can not be inverted, we move it into  $\Gamma^-$ .

$$\frac{\Gamma^- , A \supset B ; \Omega \xrightarrow{L} C^+}{\Gamma^- ; \Omega \cdot (A \supset B) \xrightarrow{L} C^+} \text{shift}_{\supset}$$

**Search.** The proof process described so far is deterministic and either succeeds finitely with a deduction, or we finally have to make a decision we might regret. Such decisions become necessary when the ordered context has become empty (marked  $\cdot$ ). At this point either identity or one of the  $\vee R$

or  $\supset L$  rules must be tried.

$$\begin{array}{c}
 \frac{P \in \Gamma}{\Gamma ; \cdot \xrightarrow{L} P} \text{id} \\
 \\ 
 \frac{\Gamma^- ; \cdot \xrightarrow{R} A}{\Gamma^- ; \cdot \xrightarrow{L} A \vee B} \vee R_1 \quad \frac{\Gamma^- ; \cdot \xrightarrow{R} B}{\Gamma^- ; \cdot \xrightarrow{L} A \vee B} \vee R_2 \\
 \\ 
 \frac{\Gamma^- , A \supset B ; \cdot \xrightarrow{R} A \quad \Gamma^- ; B \xrightarrow{L} C^+}{\Gamma^- , A \supset B ; \cdot \xrightarrow{L} C^+} \supset L
 \end{array}$$

After making a choice, we go back to a phase of inversion, either on the right (in the first premise or only premise) or on the left (in the second premise of  $\supset L$ ). Right inversion is the appropriate phase for  $\vee R_1$ ,  $\vee R_2$  and the first premise of  $\supset L$ , since the resulting formula  $A$  or  $B$ , respectively, might very well have an invertible connective so should be handled with the deterministic search first. For the second premise of  $\supset L$ , right inversion would be pointless, because its succedent  $C^+$  is already known to have a non-invertible connective. Finally observe how all inversion rules make some progress to simplify the sequents, which, in the propositional setting, can happen only finitely often.

Again, it is easy to see that the inversion calculus is sound, since it is a further restriction on the rules from the sequent calculus. It is more difficult to see that it is complete. We will not carry out this proof, but just mention that it revolves around the invertibility of the rules excepting only  $\vee R_1$ ,  $\vee R_2$ , and  $\supset L$ .

The inversion calculus is a big step forward, but it does not solve the problem with the left rule for implication, where the principal formula is copied to the first premise. We will address this in the next lecture with a so-called *contraction-free* calculus.

### 3 Soundness and Completeness of Inversion

We define the translation between ordered and unordered contexts via

$$\frac{\overline{\cdot} = \cdot}{\overline{\Omega \cdot A} = \overline{\Omega}, A}$$

Then the soundness theorem states

1. If  $\Gamma^- ; \Omega \xrightarrow{R} A$  then  $\Gamma^- , \bar{\Omega} \longrightarrow A$ , and
2. if  $\Gamma^- ; \Omega \xrightarrow{L} C^+$  then  $\Gamma^- , \bar{\Omega} \longrightarrow C^+$

The proof is straightforward by induction over the structure of the given sequent deduction. This is because the new rules just distinguish and limit certain inferences, but the otherwise the rules remain intact.

The completeness is a much more complex theorem. What we want is

1. If  $\Gamma^- , \bar{\Omega} \Longrightarrow A$  then  $\Gamma^- ; \Omega \xrightarrow{R} A$ , and
2. if  $\Gamma^- , \bar{\Omega} \Longrightarrow C^+$  then  $\Gamma^- ; \Omega \xrightarrow{L} C^+$ .

The key to this property, as for many completeness theorems, is the admissibility of cut and identity in the more restricted system. Both of these are significantly more complicated than for ordinary sequent calculus. Simple properties, such as weakening, no longer hold in the strong form we had earlier. For example, we might have

$$\frac{}{\Gamma^- ; \Omega \cdot \perp \xrightarrow{L} C^+} \perp L$$

but if we weaken, for example, as

$$\Gamma^- ; \Omega \cdot \perp \cdot ((A \vee B) \wedge (C \vee D)) \xrightarrow{L} C^+$$

we are now *forced* to break down  $(A \vee B) \wedge (C \vee D)$  completely before we can apply  $\perp L$  in each branch.

We do not replicate the proof here, but the interested reader is referred to Rob Simmons' elegant solution for an even more restricted system [Sim14].

## 4 The Contraction-Free Sequent Calculus, Revisited

At this point we need to reexamine the question from last lecture: where do we really need to make choices in this sequent calculus? We ask the question slightly differently this time, although the primary tool will still be the invertibility of rules. The question we want to ask this time: if we consider a formula on the right or on the left, can we always apply the corresponding rule without considering other choices? The difference between the two questions becomes clear, for example, in the  $P \supset L$  rule.

$$\frac{P \in \Gamma \quad \Gamma, B \longrightarrow C}{\Gamma, P \supset B \longrightarrow C} P \supset L$$

This rule is clearly invertible, because  $P \wedge (P \supset B) \equiv P \wedge B$ . Nevertheless, when we consider  $P \supset B$  we cannot necessarily apply this rule because  $P$  may not be in the remaining context  $\Gamma$ . It might become available in the context later, though, after decomposing  $\Gamma$ . So we may have to wait with applying  $P \supset L$  until  $P \in \Gamma$ .

Formulas whose left or right rules can always be applied are called left or right *asynchronous*, respectively, otherwise *synchronous*, because we may have to wait until they can be applied. We can see by examining the rules and considering the equivalences above and the methods from the last lecture, that the following formulas are asynchronous:

$$\begin{array}{ll} \text{Right asynchronous} & A \wedge B, \top, A \supset B \\ \text{Left asynchronous} & A \wedge B, \top, A \vee B, \perp, \\ & (A_1 \wedge A_2) \supset B, \top \supset B, (A_1 \vee A_2) \supset B, \perp \supset B \end{array}$$

This leaves

$$\begin{array}{ll} \text{Right synchronous} & P, A \vee B, \perp \\ \text{Left synchronous} & P, P \supset B, (A_1 \supset A_2) \supset B \end{array}$$

Atomic propositions are synchronous, because we may have to wait until it shows up in the antecedent and succedent. Disjunction is right synchronous because of the honest choice that  $\vee R_1$  versus  $\vee R_2$  imposes. Falsum is right synchronous because it needs to wait for  $\perp$  to appear in the antecedent (no  $\perp R$  rule). Atomic implication  $P \supset B$  is left synchronous, because its rule  $P \supset L$  waits for a  $P \in \Gamma$ . Nested implication  $(A_1 \supset A_2) \supset B$  could be considered left synchronous in the sense of waiting, because it is useful to handle the remaining context  $\Gamma$  before applying  $\supset \supset L$ , because any rules on  $\Gamma$  would otherwise have to be repeated in the first and second premise. But that is not actual reason! Nested implication  $(A_1 \supset A_2) \supset B$  is left synchronous, because  $\supset \supset L$  is not invertible. In its first premise, rule  $\supset \supset L$  sets out to prove  $E$  from some assumptions, which may be unsuccessful, e.g., if  $C$  is a disjunction  $P \vee Q$  for which the other synchronous cases  $\vee R_1$  or  $\vee R_2$  succeed without expanding  $(A_1 \supset A_2) \supset B$  in the antecedent.

Similar to the idea behind the inversion calculus from the previous lecture, proof search proceeds in phases. Proof search begins by breaking down all asynchronous formulas, leaving us with a situation where we have a synchronous formula on the right and only synchronous formulas on the left. We now check if id or  $P \supset L$  can be applied and use them if possible. Since these rules are invertible, this, fortunately, does not require a choice. But, of course, if they do not apply, we have to check again later as

more facts became available in  $\Gamma$ . When no more of these rules are applicable, we have to choose between  $\vee R_1$ ,  $\vee R_2$  or  $\supset\supset L$ , if the opportunity exists; if not we fail and backtrack to the most recent choice point. This makes intuitive sense. If we have a disjunction on the right and an implication with an implicational assumption on the left, there is a tradeoff of whether proof search should try proving the assumption via  $\supset\supset L$  or try proving one of the two disjuncts by  $\vee R_1$  or  $\vee R_2$ .

This strategy is complete and efficient for many typical examples, although in the end we cannot overcome the polynomial-space completeness of the intuitionistic propositional logic [Sta79]. Indeed, the search will only ever keep strictly smaller subformulas of the input (in the well-founded order) in the sequents. But we need to search through different choices to find the right combination of  $\vee R_1$ ,  $\vee R_2$  or  $\supset\supset L$  that yield a proof.

The metatheory of the contraction-free sequent calculus has been investigated separately from its use as a decision procedure by Dyckhoff and Negri [DN00]. The properties there could pave the way for further efficiency improvements by logical considerations, specifically in the treatment of atoms.

An entirely different approach to theorem proving in intuitionistic propositional logic is to use the *inverse method* [MP08] which is, generally speaking, more efficient on difficult problems, but not as direct on easier problems. We will discuss this technique in a later lecture.

Finally observe a computational interpretation of the identity theorem that  $A \rightarrow A$ . In particular, in combination with the weakening theorem,  $\Gamma, A \rightarrow A$ , which is in direct competition with the id rule which is of the same form but only applicable if  $A$  is an atomic formula. The pragmatics for proof search is that a check for applicability of the identity theorem would lead to frequent formula comparisons of complexity linear in the size of the formulas. In comparison, id is simpler because it is a direct comparison of atoms, so can essentially be made in constant time for a finite number of atoms. The identity theorem shows that it is sufficient to wait for only atomic formulas to be compared in the application of the identity rule.

## References

- [DN00] Roy Dyckhoff and Sara Negri. Admissibility of structural rules for contraction-free systems of intuitionistic logic. *Journal of Symbolic Logic*, 65:1499–1518, 2000.

- [MP08] Sean McLaughlin and Frank Pfenning. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In I.Cervesato, H.Veith, and A.Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, pages 174–181, Doha, Qatar, November 2008. Springer LNCS 5330. System Description.
- [Sim14] Robert J. Simmons. Structural focalization. *Transactions on Computational Logic*, 15(3):21:1–21:33, July 2014.
- [Sta79] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.

# Lecture Notes on Certifying Theorem Provers

15-317: Constructive Logic  
Frank Pfenning

Lecture 13  
October 17, 2017

## 1 Introduction

How do we trust a theorem prover or decision procedure for a logic? Ideally, we would prove it correct (constructively, of course!) and extract the implementation from the proof. For example, for the contraction-free sequent calculus for intuitionistic propositional logic (G4ip) we could try to prove: Every sequent  $\Gamma \rightarrow_{G4ip} A$ , either has a deduction  $\mathcal{D}$  or not. This is not an easy enterprise: We have to generalize it and then we have to carry out a well-founded induction over a multiset ordering. Then we realize that the extracted decision procedure is not very useful because it does not account for invertible rules. So we write another system, which we (a) have to prove equivalent to the first one, and (b) we have to then prove decidable. Both of these are not easy, but in the end it will give us a warm and fuzzy feeling to have a definitively correct implementation. After a couple of months of hard work. Then we realize another optimization can make our decision procedure more efficient and we have to go back to the drawing board with our proof.

In this lecture we develop an alternative approach. In this approach we first write a small, trusted *proof checker* for the logic at hand. Ideally we use the simplest and most fundamental formulation of the logic (that usually means natural deduction) to keep this checker as clean, short, and simple as possible. Proofs are fundamentally related to programs, and propositions to types, so this is the same as writing a type checker for a small core programming language.

As a second step we instrument our decision procedure or theorem prover to not just answer “yes” or “no”, but produce a proof term in case the answer is “yes”. This can then be independently checked by our small, trusted checker. Only if the checker also says “yes” do we accept and answer of the decision procedure. This does not guarantee that the prover is correct. It might, for example, incorrectly say “yes” on other propositions and supply an incorrect proof term. It could also say “no” even though the proposition is provable and we may never discover it. But, generally speaking, we are more interested in theorems and proofs, so having our prover certify theorems is a big step forward.

In this lecture we develop this in two steps: first the proof checker and then we instrument the sequent calculus to produce proof terms.

## 2 Designing a Proof Checker

Natural deduction is the simplest and purest form of logic specification, using only one judgment (*A true*) and the notion of hypothetical judgment. However, it is not ideal if we try to develop a checker for proof terms. For example, consider the rule

$$\frac{\Delta \vdash M : A}{\Delta \vdash \text{inl } M : A \vee B} \vee I_1$$

The term  $\text{inl } M$  in this rule has some inherent ambiguity because it serves as a proof term for  $A \vee B$  for any  $B$ ! In order to enforce uniqueness, we would have to annotate the constructor with the actual proposition  $B$ , for example:

$$\frac{\Delta \vdash \text{inl}^B M : A \vee B}{\Delta \vdash M : A} \vee I_1$$

We would soon find that our proof terms are littered with proposition, which is annoying and can also be inefficient in terms of space needed to represent proofs and time for checking them.

Now we should all remember the notion of *verification*. Recall that verifications proceed with introduction rules from below and elimination rules from above. When viewed in these two directions, all rules just break down the proposition we are trying to prove into its constituents. Our motivation was foundational: the meaning of a proposition should depend only on its constituents. But we can now reap the benefits in terms of proof checking: a proof terms that we extract from a verification should not need any internal propositions!

Let's start with conjunction, always a good and easy place to start.

$$\frac{A \uparrow \quad B \uparrow}{A \wedge B \uparrow} \wedge I \quad \frac{M : A \uparrow \quad N : B \uparrow}{(M, N) : A \wedge B \uparrow} \wedge I$$

This works perfectly: to check the term  $(M, N)$  against  $A \wedge B$  we can check  $M$  against  $A$  and  $N$  against  $B$ .

The elimination rules, however, do not work like that.

$$\begin{array}{c} \frac{A \wedge B \downarrow}{A \downarrow} \wedge E_1 \quad \frac{A \wedge B \downarrow}{B \downarrow} \wedge E_2 \\[10pt] \frac{R : A \wedge B \downarrow}{\text{fst } R : A \downarrow} \wedge E_1 \quad \frac{R : A \wedge B \downarrow}{\text{snd } R : B \downarrow} \wedge E_2 \end{array}$$

We *can not* check  $\text{fst } R$  against  $A$  by checking  $R$  against  $A \wedge B$  because we do not know  $B$ . But we are reading the rule the wrong way! In a verification the introduction rules are read bottom-up and the elimination rules are read top-down.

Read top-down, the  $\wedge E_1$  expresses the following: “*If  $R$  has type  $A \wedge B$  then  $\text{fst } R$  will have type  $A$ .*” Fortunately, this is entirely sensible (although at the moment we can't be sure).

Distinguishing the two judgments, we say that verifications  $A \uparrow$  are annotated with *checkable terms*  $N$ , and propositions whose use is justified with  $A \downarrow$  are annotated with *synthesizing terms*  $R$ . We are shooting for the following theorem (to be refined later, see Theorem 1):

- (i) Given  $N$  and  $A$ , either  $N : A \uparrow$  or not, and
- (ii) given  $R$  there exists an  $A$  such that  $R : A$  other there exists no such  $A$ .

We say that  $N$  *checks against*  $A$  and  $R$  *synthesizes*  $A$ .

So far we have

$$\begin{array}{lll} \text{Checkable terms} & M, N & ::= (M, N) \mid \dots \\ \text{Synthesizing terms} & R & ::= \text{fst } R \mid \text{snd } R \mid \dots \end{array}$$

Continuing with implication:

$$\frac{\overline{A \downarrow} \quad u}{\overline{\vdots} \quad u} \quad \frac{\overline{u : A \downarrow} \quad u}{\overline{\vdots} \quad u} \\ \frac{B \uparrow}{A \supset B \uparrow} \supset I^u \quad \frac{M : B \uparrow}{(\text{fn } u \Rightarrow M) : A \supset B \uparrow} \supset I^u$$

which means that functions are checkable while variables are synthesizing. How about the elimination rule? Just based on the directions of the inferences in verifications, we get the following:

$$\frac{A \supset B \downarrow \quad A \uparrow}{B \downarrow} \supset E \quad \frac{R : A \supset B \downarrow \quad M : A \uparrow}{RM : B \downarrow} \supset E$$

Recall that working from below and above meets at the  $\downarrow\uparrow$  rule. We model this by allowing and synthesizing term  $R$  as a checkable one. Intuitively, this should be okay because we can synthesize the type of  $R$  and compare it to the given type.

$$\frac{A \downarrow}{A \uparrow} \downarrow\uparrow \quad \frac{R : A \downarrow}{R : A \uparrow} \downarrow\uparrow$$

Filling in more details in our picture, we now have:

$$\begin{array}{ll} \text{Checkable terms} & M, N ::= (M, N) \mid (\text{fn } u \Rightarrow M) \mid R \mid \dots \\ \text{Synthesizing terms} & R ::= \text{fst } R \mid \text{snd } R \mid u \mid RM \mid \dots \end{array}$$

The other connectives don't present any more new and interesting ideas. We do note, for example, that  $\text{inl } M$  ends up as being a *checkable term*, which avoids the problem we encountered for natural deduction in general. In particular, we don't need to annotate  $\text{inl}$  with a type, because  $\text{inl } M$  is always *checked against*  $A \vee B$ .

$$\begin{array}{c} \frac{}{() : \top \uparrow} \top I \quad \text{no } \top E \\ \frac{M : A \uparrow}{\text{inl } M : A \vee B \uparrow} \vee I_1 \quad \frac{N : B \uparrow}{\text{inr } N : A \vee B \uparrow} \vee I_2 \\ \\ \frac{\begin{array}{c} \frac{u : A \downarrow \quad v : B \downarrow}{\vdots \quad \vdots} \\ R : A \vee B \downarrow \quad M : C \uparrow \quad N : C \uparrow \end{array}}{(\text{case } R \text{ of inl } u \Rightarrow M \mid \text{inr } v \Rightarrow N) : C \uparrow} \vee E \\ \\ \frac{\begin{array}{c} \text{no } \perp I \\ R : \perp \downarrow \end{array}}{\text{abort } R : C \uparrow} \perp E \end{array}$$

In summary (so far):

$$\begin{array}{lcl} \text{Checkable terms} & M, N ::= & (M, N) \mid (\text{fn } u \Rightarrow M) \mid R \\ & & \mid \text{inl } M \mid \text{inr } N \mid (\text{case } R \text{ of inl } u \Rightarrow M \mid v \Rightarrow N) \\ & & \mid \text{abort } R \end{array}$$

$$\text{Synthesizing terms } R ::= \text{fst } R \mid \text{snd } R \mid u \mid R M \mid \dots$$

In order to formulate our theorem, we make the hypothetical judgment explicit. We write  $\Delta = (u_1:A_1\downarrow, \dots, u_n:A_n\downarrow)$ . In lecture we worked our way up to this theorem, but I hope we have enough intuition at this point we can state it directly. We refer to this as *bidirectional type checking* because we interleave checking (bottom-up) and synthesis (top-down).

### Theorem 1 (Decidability of bidirectional type checking)

- (i) Given  $\Delta$ ,  $M$ , and  $A$ , either  $\Delta \vdash M : A \uparrow$  or  $\Delta \not\vdash M : A \uparrow$ , and
- (ii) Given  $\Delta$  and  $R$ , either there exists a unique  $A$  such that  $\Delta \vdash R : A \downarrow$  or there exists no  $A$  such that  $\Delta \vdash R : A \downarrow$ .

**Proof:** By mutual induction on the structure of  $N$  and  $R$ . For part (i) alone, we may have been able to use the structure of  $A$ , but for part (ii) we do not have  $A$ .  $\Delta$  does not give us much structure to work with, which leaves the structure of  $N$  and  $R$ .

There is subtle point in the case for  $\downarrow\uparrow$  in that the term  $R$  does not become smaller, so we also have specify that (ii) < (i). This means in an appeal to the induction hypothesis (ii) in a case for (i) the proof term can remain the same, but in an appeal to (i) from (ii), the proof term must become strictly smaller (which, fortunately, it does in all the cases).

We show four cases.

**Case:**  $M = (\text{fn } u \Rightarrow M_2)$  for some  $M_2$ . Then we distinguish cases on  $A$ .

We refer to *inversion* when a judgment could have been derived by no rule (and therefore does not hold) or just one rule (and therefore the premise would have to hold).

**Subcase:**  $A = A_1 \supset A_2$  for some  $A_1$  and  $A_2$ . Then

Either $\Delta, u:A_1\downarrow \vdash M_2 : A_2 \uparrow$ or not	by i.h.(i) on $M_2$
$\Delta, u:A_1\downarrow \vdash M_2 : A_2 \uparrow$	first subsubcase
$\Delta \vdash \text{fn } u \Rightarrow M_2 : A_1 \supset A_2 \uparrow$	by rule $\supset I$
$\Delta, u:A_1\downarrow \not\vdash M_2 : A_2 \uparrow$	second subsubcase
$\Delta \not\vdash (\text{fn } u \Rightarrow M_2 : (A_1 \supset A_2)) \uparrow$	by inversion

**Subcase:**  $A \neq A_1 \supset A_2$  for all  $A_1$  and  $A_2$ . Then

$$\Delta \not\vdash (\text{fn } u \Rightarrow M_2) : A \quad \text{by inversion}$$

**Case:**  $M = R N$  for some  $R$  and  $N$ .

$$\begin{aligned}
 & \Delta \vdash R : B \downarrow \text{ for a unique } B \text{ or there is no such } B && \text{by i.h.(ii) on } R \\
 & \Delta \vdash R : B \downarrow \text{ for a unique } B && \text{first subcase} \\
 & B = B_1 \supset B_2 \text{ for some } B_1 \text{ and } B_2 && \text{first subsubcase} \\
 & \Delta \vdash N : B_1 \uparrow \text{ or } \Delta \not\vdash N : B_1 \uparrow && \text{by i.h.(i) on } N \\
 & \Delta \vdash N : B_1 \uparrow && \text{first sub}^3\text{case} \\
 & \Delta \vdash R N : B_2 \downarrow && \text{by rule } \supset E \\
 & B_2 \text{ is unique} && \text{by inversion and uniqueness of } B \\
 & \Delta \not\vdash N : B_1 \uparrow && \text{second sub}^3\text{case} \\
 & \Delta \not\vdash R N : A \downarrow \text{ for any } A && \text{by inversion and uniqueness of } B \\
 & B \neq B_1 \supset B_2 \text{ for any } B_1 \text{ and } B_2 && \text{second subsubcase} \\
 & \Delta \not\vdash R N : A \downarrow \text{ for any } A && \text{by inversion and uniqueness of } B
 \end{aligned}$$

**Case:**  $M = R$ .

$$\begin{aligned}
 & \Delta \vdash R : A' \downarrow \text{ for a unique } A' \text{ or there is no such } A' && \text{by i.h.(ii) on } R \\
 & \text{Either } A = A' \text{ or } A \neq A' && \text{by decidability of equality on propositions} \\
 & A = A' && \text{first subcase} \\
 & \Delta \vdash R : A \uparrow && \text{by rule } \uparrow\downarrow \\
 & A \neq A' && \text{second subcase} \\
 & \Delta \not\vdash R : A \uparrow && \text{by inversion and uniqueness of } A'.
 \end{aligned}$$

**Case:**  $R = u$

$$\begin{aligned}
 & \Delta \vdash u : A \downarrow \text{ iff } u : A \in \Delta && \text{by hypothetical judgment} \\
 & A \text{ is unique} && \text{because declarations } u:A \text{ in } \Delta \text{ are unique}
 \end{aligned}$$

□

From this proof (if completed), we can extract two functions of the following types in ML:

```

check : (var * prop) list -> chk_term -> prop -> bool
synth : (var * prop) list -> syn_term -> prop option
  
```

Here, the checkable terms have type `chk_term`, synthesizing terms have type `syn_term`, and propositions are represented in the type `prop`. A context represented a list of pairs of variables and their types. There are a lot of cases to consider, but exploiting the pattern-matching facilities in ML it remains small and manageable. In a realistic implementation, one would want to print error messages or return error code instead of just `false` (for `check`) and `NONE` (for `synth`), but this is a extra-logical refinement.

As an aside, in such representation of terms we can not just include every synthesizable term as a checkable term, but we would need an explicit constructor that creates a checkable term from a synthesizable term. Such a constructor makes it less intuitive to write terms, so we can instead just have a single type of `term` and have the `check` and `synth` functions sort out which must be which.

### 3 Instrumenting a Theorem Prover

The next step will be to instrument some theorem prover so it can produce a proof term in case it succeeds. What helps us here is that (a) we already designed the sequent calculus as a purely bottom-up system for searching for a verification, and (b) more efficient search procedures (such as G4ip, which is in fact a decision procedure) are actually presented as refinements of the sequent calculus. As an example we consider here G4, which is the sequent calculus from [Lecture 11, Section 4](#).

$$\begin{array}{c}
\overline{\Gamma, P \longrightarrow P} \text{ id} \\
\frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B} \wedge R \quad \frac{\Gamma, A, B \longrightarrow C}{\Gamma, A \wedge B \longrightarrow C} \wedge L \\
\frac{}{\Gamma \longrightarrow \top} \top R \quad \frac{\Gamma \longrightarrow C}{\Gamma, \top \longrightarrow C} \top L \\
\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B} \vee R_1 \quad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B} \vee R_2 \quad \frac{\Gamma, A \longrightarrow C \quad \Gamma, B \longrightarrow C}{\Gamma, A \vee B \longrightarrow C} \vee L \\
\text{no } \perp R \text{ rule} \quad \overline{\Gamma, \perp \longrightarrow C} \perp L \\
\frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B} \supset R \quad \frac{\Gamma, A \supset B \longrightarrow A \quad \Gamma, B \longrightarrow C}{\Gamma, A \supset B \longrightarrow C} \supset L
\end{array}$$

Recall that this is an optimization of the system which we obtained from translating

$$\begin{array}{ccc}
A_1 \downarrow, \dots, A_n \downarrow & & \\
\vdots & & \vdots \\
C \uparrow & \text{to} & A_1, \dots, A_n \Rightarrow C
\end{array}$$

The assignment of synthesizing and checking terms to the verifications on the side of natural deduction suggests the corresponding annotations on the side of sequents.

$$\begin{array}{ccc}
R_1 : A_1 \downarrow, \dots, R_n : A_n \downarrow & & \\
\vdots & & \vdots \\
N : C \uparrow & \text{to} & R_1 : A_1, \dots, R_n : A_n \Rightarrow N : C
\end{array}$$

Furthermore, since propositions  $A \downarrow$ , working downwards from hypotheses, are always fully justified as we are searching for a proof while  $C \uparrow$  is unknown until we complete the proof, the theorem we are aiming for is:

### Theorem 2 (Sequent Proof Annotation)

*For every deduction  $A_1, \dots, A_n \longrightarrow C$  and for all hypotheses  $\Delta$  with  $\Delta \vdash R_1 : A_1 \downarrow, \dots, \Delta \vdash R_n : A_n \downarrow$  there exists an  $N$  such that  $R_1 : A_1, \dots, R_n : A_n \longrightarrow N : C$  and  $\Delta \vdash N : C \uparrow$*

**Proof:** By induction on the structure of  $A_1, \dots, A_n \rightarrow C$   $\square$

Rather than show the proof case by case, we develop the proof term annotation case by case. Let's start with

$$\frac{\Gamma, A \rightarrow B}{\Gamma \rightarrow A \supset B} \supset R$$

We can annotate the antecedents in  $\Gamma$  with a sequence  $\rho = (R_1, \dots, R_n)$  of synthesizing terms, which we abbreviate by  $\rho : \Gamma$ . By induction hypothesis, and with a fresh variable  $u$ , we get some  $N$  such that

$$\rho : \Gamma, u : A \rightarrow N : B$$

from which we can glean that the annotated rule should be

$$\rho : \Gamma \rightarrow (\text{fn } u \Rightarrow N) : A \supset B$$

Moreover, since  $\Delta, u : A \downarrow \vdash u : A \downarrow$ , we have  $\Delta, u : A \downarrow \vdash N : B \uparrow$  and hence  $\Delta \vdash (\text{fn } u \Rightarrow N) : A \supset B \downarrow$  by  $\supset I$ .

This means our annotated rule should be

$$\frac{\rho : \Gamma, u : A \rightarrow N : B}{\rho : \Gamma \rightarrow (\text{fn } u \Rightarrow N) : A \supset B} \supset R$$

As a second case, we consider  $\supset L$ . We have

$$\frac{\Gamma, A \supset B \rightarrow A \quad \Gamma, B \rightarrow C}{\Gamma, A \supset B \rightarrow C} \supset L$$

We are also given  $\rho : \Gamma, R : A \supset B$ . By induction hypothesis, we get an  $M$  such that

$$\rho : \Gamma, R : A \supset B \rightarrow M : A \quad \text{and} \quad \Delta \vdash M : A$$

In order to be able to apply the induction hypothesis to the second premise, we need some (synthesizing) term, denoting a proof of  $B$ . We we have  $R : A \supset B$  and  $M : A$ , so  $R M : B$  and we obtain from the induction hypothesis some  $N$  such that

$$\rho : \Gamma, R N : B \rightarrow N : C \quad \text{and} \quad \Delta \vdash N : C$$

which allows us to choose  $N : C$  in the conclusion as well. Summarizing this as a rule, we get

$$\frac{\rho : \Gamma, R : A \supset B \longrightarrow M : A \quad \rho : \Gamma, RM : B \longrightarrow N : C}{\rho : \Gamma, R : A \supset B \longrightarrow N : C} \supset L$$

As a final case, we consider the identity rule.

$$\overline{\Gamma, P \longrightarrow P} \text{ id}$$

We also have  $\rho : \Gamma, R : P$  so we can choose  $N = R$ :

$$\overline{\rho : \Gamma, R : P \longrightarrow R : P} \text{ id}$$

and  $\Delta \vdash R : P \downarrow$  implies  $\Delta \vdash R : P \uparrow$  by rule  $\downarrow\uparrow$ .

We now summarize all the rules, but reusing the notation  $\Gamma$  for  $\rho : \Gamma$  to make the rules more readable.

$$\begin{array}{c} \overline{\Gamma, R : P \longrightarrow R : P} \text{ id} \\[10pt] \frac{\Gamma \longrightarrow M : A \quad \Gamma \longrightarrow N : B}{\Gamma \longrightarrow (M, N) : A \wedge B} \wedge R \qquad \frac{\Gamma, \text{fst } R : A, \text{snd } R : B \longrightarrow N : C}{\Gamma, R : A \wedge B \longrightarrow N : C} \wedge L \\[10pt] \frac{}{\Gamma \longrightarrow () : \top} \top R \qquad \frac{\Gamma \longrightarrow C}{\Gamma, R : \top \longrightarrow C} \top L \\[10pt] \frac{\Gamma \longrightarrow M : A}{\Gamma \longrightarrow \text{inl } M : A \vee B} \vee R_1 \qquad \frac{\Gamma \longrightarrow N : B}{\Gamma \longrightarrow \text{inr } N : A \vee B} \vee R_2 \\[10pt] \frac{\Gamma, u : A \longrightarrow N_1 : C \quad \Gamma, v : B \longrightarrow N_2 : C}{\Gamma, R : A \vee B \longrightarrow (\text{case } R \text{ of inl } u \Rightarrow N_1 \mid \text{inr } v \Rightarrow N_2) : C} \vee L \\[10pt] \text{no } \perp R \text{ rule} \qquad \overline{\Gamma, R : \perp \longrightarrow \text{abort } R : C} \perp L \\[10pt] \frac{\Gamma, u : A \longrightarrow N : B}{\Gamma \longrightarrow (\text{fn } u \Rightarrow N) : A \supset B} \supset R \\[10pt] \frac{\Gamma, R : A \supset B \longrightarrow M : A \quad \Gamma, (RM) : B \longrightarrow N : C}{\Gamma, R : A \supset B \longrightarrow N : C} \supset L \end{array}$$

## 4 Justifying Cut

In homework assignment 6 you are asked to provide a similar proof term assignment in G4ip, which you previously implemented in homework 5. This requires one further thought: how would we handle the rule of cut if it were needed? Let's come back to our inductive proof of Theorem 2 and consider that case that we would have cut as a rule (but we write it as the admissible rule it is):

$$\frac{\Gamma \longrightarrow A \quad \Gamma, A \longrightarrow C}{\Gamma \longrightarrow C} \text{cut}$$

We know, by the fact that cut is admissible, that there is a cut-free proof of the conclusion. We could construct this using the proof of admissibility (which was constructive) and annotate the result. Unfortunately, the result could be quite large, since cut elimination can explode the size of the proof.

Alternatively, we can make up a new kind of proof term for this rule, standing in for what cut elimination might compute. First, since we have  $\rho : \Gamma$  we can appeal to the induction hypothesis and construct and  $M$  such that

$$\rho : \Gamma \longrightarrow M : A$$

If we could only turn the checkable term  $M$  into a synthesizing term  $R$ , we could use this to justify the antecedent  $A$  in the second premise. For this we create a new construct  $(M : A)$  in the syntax for synthesizing terms. It synthesizes  $A$  (which is therefore unique) if  $M$  checks against  $A$ . But  $M$  (which we obtained from an appeal to the induction hypothesis) was a checkable term! Then, again by induction hypothesis we obtain

$$\rho : \Gamma, (M : A) : A \longrightarrow N : C$$

which we can use to conclude

$$\rho : \Gamma \longrightarrow N : C$$

as required. This leads to the rule

$$\frac{\Gamma \longrightarrow M : A \quad \Gamma, (M : A) : A \longrightarrow N : C}{\Gamma \longrightarrow N : C} \text{cut}$$

We would need the new rule

$$\frac{\Delta \vdash M : A \uparrow}{\Delta \vdash (M : A) : A \downarrow} \uparrow\downarrow$$

For verifications, this cannot be a primitive rule (since it destroys the meaning explanation for the connectives), but we can use it if we extend our syntax with  $(M : A)$  as a new form of synthesizing term.

Alternatively, we could use the let form by justifying  $A$  by a variable  $u$  which is discharged using the verification of  $A$  in the conclusion:

$$\frac{\Gamma \vdash M : A \quad \Gamma, u : A \vdash N : C}{\Gamma \vdash (\text{let } u : A = M \text{ in } N) : C} \text{ cut}$$

The let form here is necessary in the concluding because otherwise the conclusion would still depend on  $u$ . This form is much more pleasant from a programming perspective. It also means we can type every term (not just normal terms) if we annotate the let form with its type. Additionally, this is the only form where we need a type. In some ways, this is the essence of bidirectional type-checking: only redexes need to be annotated with a type. If all redexes are expressed as let forms, this means only let forms need to be annotated, and really only if the term we are assigning is only checkable. If it were synthesizing, as in  $\text{let } u = R \text{ in } N$ , we could synthesize the type  $A$  of  $R$  and proceed to check  $N$  under the antecedent  $u : A$ .

Adding both of these alternatives to the syntax (even though only one is really required), we obtain this syntax that allows us to express arbitrary proofs, not just those annotating verifications.

$$\begin{array}{ll} \text{Checkable terms} & M, N ::= (M, N) \mid (\text{fn } u \Rightarrow M) \mid R \\ & \mid \text{inl } M \mid \text{inr } N \mid (\text{case } R \text{ of inl } u \Rightarrow M \mid v \Rightarrow N) \\ & \mid \text{abort } R \\ & \mid (\text{let } u : A = M \text{ in } N) \end{array}$$

$$\text{Synthesizing terms } R ::= \text{fst } R \mid \text{snd } R \mid u \mid RM \mid (M : A)$$

# Lecture Notes on Logic Programming

15-317: Constructive Logic  
Frank Pfenning\*

Lecture 14  
October 19, 2017

## 1 Computation vs. Deduction

The previous lectures explored a connection between logic and computation based on the observation that once we have a (constructive) proof, it corresponds to a functional program (proofs-as-programs). In this lecture we switch to an entirely different connection between logic and computation. The starting point is that the search for a proof has a computational interpretation. We interpret logical rules as programs that are executed by proof search according to a fixed strategy. This gives rise to the *formulas-as-programs* paradigm, where we relate deductive proof search to computation in logic programming. And, in fact, this development was foreshadowed to some extent by the previous lectures on proof search.

Logic programming is a particular way to approach programming. Other paradigms we might compare it to are imperative programming or functional programming. The divisions are not always clear-cut—a functional language may also have some imperative aspects, for example—but the mindset of various paradigms is quite different and determines how we design and reason about programs.

To understand logic programming, we first examine the difference between computation and deduction. To *compute* we start from a given expression and, according to a fixed set of rules (the program) generate a result. For example,  $25 + 46 \rightarrow (2 + 4 + 1)1 \rightarrow (6 + 1)1 \rightarrow 71$  for a computation of decimal addition with carry. To *deduce* we start from a conjecture

---

\*With edits by André Platzer

and, according to a fixed set of rules (the axioms and inference rules), try to construct a proof of the conjecture. So computation is mechanical and requires no ingenuity, while deduction is a creative process. For example, for all  $n > 2$ :  $a^n + b^n \neq c^n$ , ... 357 years of hard work ..., QED.

Philosophers, mathematicians, and computer scientists have tried to unify the two, or at least to understand the relationship between them for centuries. For example, George Boole<sup>1</sup> succeeded in reducing a certain class of logical reasoning to computation in so-called Boolean algebras. Since the fundamental undecidability breakthroughs in the 20th century we know that not everything we can reason about is in fact mechanically computable, even if we follow a well-defined fixed set of formal rules.

Yet even so, we should find a striking similarity of the above descriptions of computation and deduction. Both start from some initial input and follow a fixed set of rules, whether program or axioms and inference rules.

In this course we are interested in a connection of a different kind. A first observation is that computation can be seen as a limited form of deduction, because computation actually establishes theorems, too. For example,  $25 + 46 = 71$  is both the result of a computation, and a theorem of arithmetic. Conversely, deduction can be considered a form of computation if only we fix a strategy for proof search, removing the guesswork (and the possibility of employing ingenuity!) from the deductive process.

This latter idea is the foundation of logic programming. *Logic program computation proceeds by proof search according to a fixed strategy.* By knowing what this strategy is, we can implement particular algorithms in logic, and execute the algorithms by proof search according to this fixed strategy.

## 2 Judgments and Proofs

Since logic programming computation is proof search, to study logic programming means to study proofs. We adopt here the approach by Martin-Löf [3]. Although he studied logic as a basis for functional programming rather than logic programming, his ideas are more fundamental and therefore equally applicable in both paradigms.

Recall the most basic notion is that of a *judgment*, which is an object of knowledge. We know a judgment because we have evidence for it. The kind of evidence we are most interested in is a *proof*, which we display as a

---

<sup>1</sup>1815–1864

*deduction using inference rules* in the form

$$\frac{J_1 \dots J_n}{J} R$$

where  $R$  is the name of the rule (often omitted),  $J$  is the judgment established by the inference (the *conclusion*), and  $J_1, \dots, J_n$  are the *premisses* of the rule. We can read it as

*If  $J_1$  and  $\dots$  and  $J_n$  then we can conclude  $J$  by virtue of rule  $R$ .*

By far the most common judgment is the truth of a proposition  $A$ , which we write as  $A$  *true*. Because we will be occupied almost exclusively with the truth of propositions for quite some time in this course, and have now mastered the nuances of separating a proposition from a judgment about it, we will from now on omit the trailing “*true*” and just write  $A$  in a rule, when really we still mean  $A$  *true*.

To give some simple examples we need a language to express propositions. We start with *terms*  $t$  that have the form  $f(t_1, \dots, t_n)$  where  $f$  is a *function symbol* of arity<sup>2</sup>  $n$  and  $t_1, \dots, t_n$  are the arguments. Terms can have variables in them, which we generally denote by upper-case letters in the context of logic programming. *Atomic propositions*  $P$  have the form  $p(t_1, \dots, t_n)$  where  $p$  is a *predicate symbol* of arity  $n$  and  $t_1, \dots, t_n$  are its arguments. Later we will introduce more general forms of propositions, built up by logical connectives and quantifiers from atomic propositions.

In our first set of examples we represent natural numbers  $0, 1, 2, \dots$  as terms of the form  $0, s(0), s(s(0)), \dots$ , using two function symbols ( $0$  of arity 0 and  $s$  of arity 1).<sup>3</sup> The first predicate we consider is the predicate even of arity 1. Its meaning is defined by two inference rules:

$$\frac{}{\text{even}(0)} \text{evz} \qquad \frac{\text{even}(N)}{\text{even}(s(s(N)))} \text{evs}$$

The first rule, evz, expresses that 0 is even. It has no premiss and therefore is like an axiom. The second rule, evs, expresses that if  $N$  is even, then  $s(s(N))$  is also even. Here,  $N$  is a *schematic variable* of the inference rule: every instance of the rule where  $N$  is replaced by a concrete term represents a valid inference. We have no more rules, so we think of these two as

---

<sup>2</sup>A function  $f$  of arity  $n$  expects the  $n$  arguments  $t_1, \dots, t_n$ .

<sup>3</sup>This unary representation is not how numbers are represented in practical logic programming languages such as Prolog, but it is a convenient source of simple examples.

completely defining the predicate even in the sense that there are no other circumstances under which we would know  $\text{even}(N)$  except those justified by a series of uses of both rules.

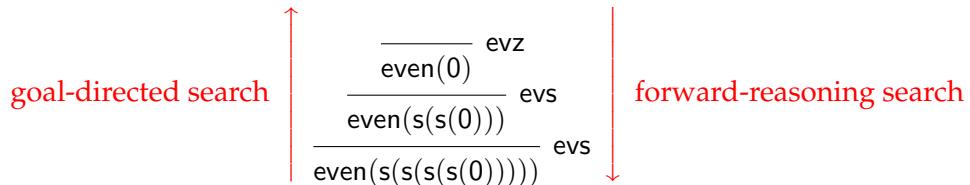
The following is a trivial example of a deduction, showing that 4 is even:

$$\frac{\frac{\frac{\overline{\quad}}{\text{even}(0)} \text{ evz}}{\text{even}(\text{s}(\text{s}(0)))} \text{ evs}}{\text{even}(\text{s}(\text{s}(\text{s}(\text{s}(0)))))} \text{ evs}$$

It used the rule evs twice: once with  $N = 0$  and once with  $N = \text{s}(\text{s}(0))$ .

### 3 Proof Search

To make the transition from inference rules to logic programming we need to impose a particular proof search strategy. Two fundamental ideas suggest themselves: we could either search backward from the conjecture, growing a (potential) proof tree upwards until all resulting premises are proved so that it turns into a proof, or we could work forwards from the axioms applying rules until we arrive at the conjecture. We call the first one *goal-directed* and the second one *forward-reasoning*.



In the logic programming literature we find the terminology *top-down* for goal-directed, and *bottom-up* for forward-reasoning, but this goes counter to the direction in which the proof tree is constructed. Logic programming was conceived with goal-directed search, and this is still the dominant direction since it underlies Prolog, the most popular logic programming language. Later in the class, we will also have an opportunity to consider forward reasoning.

**Goal-directed Proof Search.** In the first approximation, the goal-directed strategy we apply is very simple: given a conjecture (called the *goal*) we determine which inference rules might have been applied to arrive at this conclusion. We select one of them and then recursively apply our strategy to

all the premisses as subgoals. If there are no premisses we have completed the proof of the goal. Of course, we will consider many refinements and more precise descriptions of this basic idea of goal-directed proof search in this course.

For example, consider the conjecture  $\text{even}(\text{s}(\text{s}(0)))$ . We now execute the logic program consisting of the two rules  $\text{evz}$  and  $\text{evs}$  to either prove or refute this goal. We notice that the only rule with a matching conclusion is  $\text{evs}$ . Our partial proof now looks like

$$\frac{\vdots}{\text{even}(\text{s}(\text{s}(0)))} \text{ evs}$$

$$\frac{\text{even}(0)}{\text{even}(\text{s}(\text{s}(0)))} \text{ evs}$$

with  $\text{even}(0)$  as the only subgoal.

Considering the subgoal  $\text{even}(0)$  we see that this time only the rule  $\text{evz}$  could have this conclusion. Moreover, this rule has no premisses so the computation terminates successfully, having found the proof

$$\frac{\text{even}(0)}{\text{even}(\text{s}(\text{s}(0)))} \text{ evs.}$$

Actually, most logic programming languages will not show the proof in this situation, but only answer `yes` if a proof has been found, which means the conjecture was true.

**Failing Proof Search.** Now consider the goal  $\text{even}(\text{s}(\text{s}(\text{s}(0))))$ . Clearly, since 3 is not even, the computation must fail to produce a proof. Following our strategy, we first reduce this goal using the  $\text{evs}$  rule to the subgoal  $\text{even}(\text{s}(0))$ , with the incomplete proof

$$\frac{\vdots}{\text{even}(\text{s}(0))} \text{ evs.}$$

$$\frac{\vdots}{\text{even}(\text{s}(\text{s}(0)))} \text{ evs.}$$

At this point we note that there is no rule whose conclusion matches the goal  $\text{even}(\text{s}(0))$ . We say proof search *fails*, which will be reported back as the result of the computation, usually by printing `no`.

Since we think of the two rules as the complete definition of even we conclude that  $\text{even}(\text{s}(0))$  is *false*. This example illustrates *negation as failure*, which is a common technique in logic programming. Notice, however, that there is an asymmetry: in the case where the conjecture was true, search constructed an explicit proof which provides evidence for its truth. In the case where the conjecture was false, no evidence for its falsehood is immediately available since all we can say is that we tried to find a proof in all possible ways and failed in each. This means that *negation does not have first-class status in logic programming*.

## 4 Answer Substitutions

In the even example the response to a goal is either yes, in which case a proof has been found, or no, if all attempts at finding a proof fail finitely. In general, it is also possible that proof search does not terminate. But how can we write logic programs to compute values?

Since every natural number is either even or odd, the only expected answers are yes or no in that case. So let's look at an example where we actually expect a computed value as an answer. As an example we consider programs to compute sums and differences of natural numbers in the representation from the previous section. We start by specifying the underlying *relation* and then illustrate how it can be used for computation. The relation in this case is  $\text{plus}(m, n, p)$  which should hold if  $m + n = p$ . We use the recurrence

$$\begin{array}{rcl} (m + 1) + n & = & (m + n) + 1 \\ 0 + n & = & n \end{array}$$

as our guide because it counts down the first argument to 0, which will eventually happen for natural numbers. We obtain

$$\frac{\text{plus}(M, N, P)}{\text{plus}(\text{s}(M), N, \text{s}(P))} \text{ ps} \qquad \frac{}{\text{plus}(0, N, N)} \text{ pz.}$$

Now consider a goal of the form  $\text{plus}(\text{s}(0), \text{s}(0), R)$  where  $R$  is an unknown. This represents the question if there exists an  $R$  such that the relation  $\text{plus}(\text{s}(0), \text{s}(0), R)$  holds. Search not only constructs a proof, but, with some bookkeeping, also a term  $t$  for  $R$  such that  $\text{plus}(\text{s}(0), \text{s}(0), t)$  is true. This term  $t$  for  $R$  is the answer for the question whether there is a choice for  $R$  such that  $\text{plus}(\text{s}(0), \text{s}(0), R)$  holds.

For the original goal,  $\text{plus}(\text{s}(0), \text{s}(0), R)$ , only the rule ps could apply because of a mismatch between 0 and  $\text{s}(0)$  in the first argument to plus in the

conclusion. We also see that the  $R$  must have the form  $s(P)$  for some  $P$ , although we do not know yet what  $P$  should be.

$$\frac{\vdots}{\frac{\text{plus}(0, s(0), P)}{\text{plus}(s(0), s(0), R)}} \text{ ps with } R = s(P)$$

For its subgoal only the pz rule applies and we see that  $P$  must equal  $s(0)$ .

$$\frac{\text{proof search}}{\frac{\text{plus}(0, s(0), P) \text{ pz with } P = s(0)}{\text{plus}(s(0), s(0), R) \text{ ps with } R = s(P)}} \text{ substitute answers}$$

If we carry out the substitutions backwards and put  $P = s(0)$  into  $R = s(P)$  giving  $R = s(s(0))$ , we obtain the complete proof

$$\frac{\text{plus}(0, s(0), s(0)) \text{ pz}}{\text{plus}(s(0), s(0), s(s(0)))} \text{ ps}$$

which is explicit evidence that  $1 + 1 = 2$ . Instead of the full proof, implementations of logic programming languages mostly just print the substitution for the unknowns in the original goal, in this case  $R = s(s(0))$ .

Some terminology of logic programming: the original goal is called the *query*, its unknowns are *logic variables*, and the result of the computation is an *answer substitution* for the logic variables, suppressing the proof.

## 5 Backtracking

Sometimes during proof search the goal matches the conclusion of more than one rule. This is called a *choice point*. When we reach a choice point we always pick the first among the rules that match, in the order they were presented. If that attempt at a proof fails, we try the second one that matches, and so on. This process is called *backtracking*.

As an example, consider the query  $\text{plus}(M, s(0), s(s(0)))$ , intended to compute an  $m$  such that  $m + 1 = 2$ , that is,  $m = 2 - 1$ . This demonstrates that we can use the same logic program (here: the definition of the plus predicate) in different ways (before: addition, now: subtraction).

The conclusion of the rule `pz`,  $\text{plus}(0, N, N)$ , does not match because the second and third argument of the query are different. However, the rule `ps` applies and we obtain

$$\frac{\vdots}{\text{plus}(M_1, \text{s}(0), \text{s}(0))} \text{ ps with } M = \text{s}(M_1)$$

At this point both rules, `ps` and `pz`, match. We use the rule `ps` because it is listed first, leading to

$$\frac{\vdots}{\frac{\text{plus}(M_2, \text{s}(0), 0)}{\text{plus}(M_1, \text{s}(0), \text{s}(0))} \text{ ps with } M_1 = \text{s}(M_2)} \text{ ps with } M = \text{s}(M_1)$$

At this point no rule applies at all and this attempt fails. So we return to our earlier choice point and try the second alternative, `pz`.

$$\frac{\overline{\text{plus}(M_1, \text{s}(0), \text{s}(0))} \text{ pz with } M_1 = 0}{\text{plus}(M, \text{s}(0), \text{s}(s(0)))} \text{ ps with } M = \text{s}(M_1)$$

At this point the proof is complete, with the answer substitution  $M = \text{s}(0)$ .

Note that with even a tiny bit of foresight we could have avoided the failed attempt by picking the rule `pz` first. But even this small amount of ingenuity cannot be permitted: in order to have a satisfactory programming language we must follow every step prescribed by the search strategy precisely.

## 6 Modes

Wait, why did the above examples work with the same rules defining `plus`? Well,  $\text{plus}(M, N, R)$  is a relation and we can make use of such a relation by providing inputs for  $M$  and  $N$  and computing an answer for  $R$ , which adds  $M$  to  $N$ . Or we can make use of the same relation by providing inputs for  $N$  and  $R$  and computing an answer for  $M$ , which subtracts  $N$  from  $R$ . It is useful to keep track of both modes of using `plus`.

mode	meaning
$\text{plus}(+N, +M, -R)$	instantiated terms as input for first two arguments compute output for last argument (here: addition)
$\text{plus}(-N, +M, +R)$	instantiated terms as input for last two arguments compute output for first argument (here: subtraction)

Even if not enforced in Prolog, it is also good style to describe the expected types of the arguments, so you will also see  $\text{plus}(+\text{nat}, +\text{nat}, -\text{nat})$  with natural numbers  $\text{nat}$ . Finally you will also often see the notation  $\text{plus}/3$  to refer to the  $\text{plus}$  predicate with 3 arguments.

It is important to remember that these modes do not come for free, and that they are not checked in a typical logic programming implementation. This means as part of the programming process we need to carefully check that the modes work out correctly, or we risk nontermination (mostly) or incorrect answers (sometimes).

Let's reconsider the specification for addition.

$$\frac{\text{plus}(M, N, P)}{\text{plus}(\text{s}(M), N, \text{s}(P))} \text{ ps} \qquad \frac{}{\text{plus}(0, N, N)} \text{ pz}$$

To check that this is well-modeled under the mode  $\text{plus}(+, +, -)$  we need to prove by induction on the structure of the rules that if the values of the first two arguments of  $\text{plus}$  are *known* then the third argument will be *known* in case the search succeeds.

**Case:** Rule pz. We know 0 (which gives no useful information) and  $N$ . We have to show we know  $N$ , which happens to be one of our assumptions and we declare this rule to be well-modeled.

**Case:** Rule ps. We know  $\text{s}(M)$  and  $N$ . This means we also know  $M$ , and we can apply the induction hypothesis to conclude that  $P$  will be known if the search for a proof of the premise succeeds. But that means that  $\text{s}(P)$  will also be known and this rule is also mode correct.

Determining that subtraction  $\text{plus}(+, -, +)$  is a valid mode is similarly straightforward:

**Case:** Rule pz. We know  $N$  from the third argument, so we also know the second.

**Case:** Rule ps. We know  $N$  and  $\text{s}(P)$  the second and third arguments in the conclusion. This means we also know  $P$  and can apply the induction hypothesis to conclude that we know  $M$ . But that means we can construct  $\text{s}(M)$ , the first argument of the conclusion.

## 7 Subgoal Order

Another kind of choice arises when an inference rule has multiple premises, namely the order in which we try to find a proof for them. Of course, logically the order should not be relevant because the final proof is a proof no matter in which order we went to find it. But operationally the behavior of a program can be quite different.

As an example, we define  $\text{times}(m, n, p)$  which should hold if  $m \times n = p$ . We implement the recurrence

$$\begin{array}{rcl} 0 \times n & = & 0 \\ (m + 1) \times n & = & (m \times n) + n \end{array}$$

in the form of the following two inference rules.

$$\frac{}{\text{times}(0, N, 0)} \text{tz} \quad \frac{\text{times}(M, N, P) \quad \text{plus}(P, N, Q)}{\text{times}(\text{s}(M), N, Q)} \text{ts}$$

As an example we compute  $1 \times 2 = Q$ . The first step is determined.

$$\frac{\vdots \quad \vdots}{\text{times}(0, \text{s}(\text{s}(0)), P) \quad \text{plus}(P, \text{s}(\text{s}(0)), Q)} \text{ts}$$

$$\text{times}(\text{s}(0), \text{s}(\text{s}(0)), Q)$$

Now if we solve the left subgoal first, there is only one applicable rule, tz, which forces  $P = 0$

$$\frac{\text{times}(0, \text{s}(\text{s}(0)), P)}{\text{times}(\text{s}(0), \text{s}(\text{s}(0)), Q)} \text{tz } (P = 0) \quad \frac{\vdots \quad \vdots}{\text{plus}(P, \text{s}(\text{s}(0)), Q)} \text{ts}$$

Now since  $P = 0$  from the first subgoal, which we, thus, know also for the second subgoal, there is only one rule that applies to the second subgoal, too, and we obtain correctly

$$\frac{\text{times}(0, \text{s}(\text{s}(0)), P) \text{ tz } (P = 0) \quad \frac{\text{plus}(P, \text{s}(\text{s}(0)), Q)}{\text{plus}(P, \text{s}(\text{s}(0)), Q)} \text{ pz } (Q = \text{s}(\text{s}(0)))}{\text{times}(\text{s}(0), \text{s}(\text{s}(0)), Q)} \text{ ts.}$$

On the other hand, if we were to solve the right subgoal  $\text{plus}(P, \text{s}(\text{s}(0)), Q)$  first, then we would have no information on  $P$  and  $Q$ , so both rules for plus

apply. Since  $\text{ps}$  is given first, the strategy discussed in the previous section means that we try it first, which leads to

$$\frac{\begin{array}{c} \vdots \\ \text{times}(0, \text{s}(\text{s}(0)), P) \end{array} \quad \frac{\begin{array}{c} \vdots \\ \text{plus}(P_1, \text{s}(\text{s}(0)), Q_1) \\ \text{plus}(P_1, \text{s}(\text{s}(0)), Q_1) \end{array} \text{ps } (P = \text{s}(P_1), Q = \text{s}(Q_1))}{\text{plus}(P, \text{s}(\text{s}(0)), Q)} \text{ ts.}}{\text{times}(\text{s}(0), \text{s}(\text{s}(0)), Q)}$$

Again, rules  $\text{ps}$  and  $\text{ts}$  are both applicable, with  $\text{ps}$  listed first, so we continue:

$$\frac{\begin{array}{c} \vdots \\ \text{times}(0, \text{s}(\text{s}(0)), P) \end{array} \quad \frac{\begin{array}{c} \vdots \\ \text{plus}(P_2, \text{s}(\text{s}(0)), Q_2) \\ \text{plus}(P_2, \text{s}(\text{s}(0)), Q_2) \end{array} \text{ps } (P_1 = \text{s}(P_2), Q_1 = \text{s}(Q_2)) \\ \vdots \\ \text{plus}(P_1, \text{s}(\text{s}(0)), Q_1) \end{array} \text{ps } (P = \text{s}(P_1), Q = \text{s}(Q_1))}{\text{plus}(P, \text{s}(\text{s}(0)), Q)} \text{ ts.}}{\text{times}(\text{s}(0), \text{s}(\text{s}(0)), Q)}$$

It is easy to see that this will go on indefinitely, and computation will not terminate.

In fact, in light of the backtracking we observed here, we might want to reorder the rules so that  $\text{pz}$  comes before  $\text{ps}$  since  $\text{pz}$  gives short proofs. Likewise, the right premise of  $\text{ts}$  has two schema variables that are still unknown while the left premise has only one. That serves as a heuristic indication that  $\text{tz}$  might have the appropriate order. These are heuristic considerations, however, and a more detailed analysis is necessary to determine the computationally most suitable form.

This examples illustrate that the order in which subgoals are solved can have a strong impact on the computation. Here, proof search either completes in two steps or does not terminate. This is a consequence of fixing an operational reading for the rules. The standard solution is to attack the subgoals in *left-to-right order*. We observe here a common phenomenon of logic programming: two definitions, entirely equivalent from the logical point of view, can be very different operationally. Actually, this is also true for functional programming: two implementations of the same function can have very different complexity. This debunks the myth of “declarative programming”—the idea that we only need to specify the problem rather than design and implement an algorithm for its solution. However, we can assert that both specification and implementation can be expressed in

the language of logic. Furthermore, correctness is easily established (separately from the computational question of termination and efficiency) in either decision just from the rules. As we will see later when we come to logical frameworks, we can integrate even correctness proofs into the same formalism!

In lecture we now gave another example: numbers in binary form and some predicates on them. We will show these with the material of the next lecture.

## 8 Prolog Notation<sup>4</sup>

By far the most widely used logic programming language is Prolog, which actually is a family of closely related languages. There are several good textbooks, language manuals, and language implementations, both free and commercial. A good resource is the FAQ of the Prolog newsgroup<sup>5</sup>. For this course we use GNU Prolog<sup>6</sup> although the programs should run in just about any Prolog since we avoid the more advanced features.

The two-dimensional presentation of inference rules does not lend itself to a textual format. The Prolog notation for a rule

$$\frac{J_1 \dots J_n}{J} R$$

is

$$J \leftarrow J_1, \dots, J_n.$$

where the name of the rule is omitted and the left-pointing arrow is rendered as ':-' in a plain text file.

$$J :- J_1, \dots, J_n.$$

We read this as

$$J \text{ if } J_1 \text{ and } \dots \text{ and } J_n.$$

Prolog terminology for an inference rule is a *clause*, where  $J$  is the *head* of the clause and  $J_1, \dots, J_n$  is the body. Therefore, instead of saying that we "search for an inference rule whose conclusion matches the conjecture", we say that we "search for a clause whose head matches the goal".

As an example, we show the earlier programs in Prolog notation.

---

<sup>4</sup>Covered in the next lecture

<sup>5</sup><https://groups.google.com/forum/#!forum/comp.lang.prolog>

<sup>6</sup><http://www.gprolog.org/>

```

even(z).
even(s(s(N))) :- even(N).

plus(s(M), N, s(P)) :- plus(M, N, P).
plus(z, N, N).

times(z, N, z).
times(s(M), N, Q) :-
    times(M, N, P),
    plus(P, N, Q).

```

Clauses are tried in the order they are presented in the program. Subgoals are solved in the order they are presented in the body of a clause.

## 9 Unification<sup>7</sup>

One important operation during search is to determine if the conjecture matches the conclusion of an inference rule (or, in logic programming terminology, if the goal unifies with the head of a clause). This operation is a bit subtle, because the rule may contain schematic variables, and the goal may also contain logical variables.

As a simple example (which we glossed over before), consider the goal

$\text{plus}(\text{s}(0), \text{s}(0), R)$

and the clause

$\text{plus}(\text{s}(M), N, \text{s}(P)) \leftarrow \text{plus}(M, N, P)$

We need to find some way to instantiate  $M$ ,  $N$ , and  $P$  in the clause head and  $R$  in the goal such that  $\text{plus}(\text{s}(0), \text{s}(0), R) = \text{plus}(\text{s}(M), N, \text{s}(P))$ , by which we mean that  $\text{plus}(\text{s}(0), \text{s}(0), R)$  and  $\text{plus}(\text{s}(M), N, \text{s}(P))$  become syntactically identical.

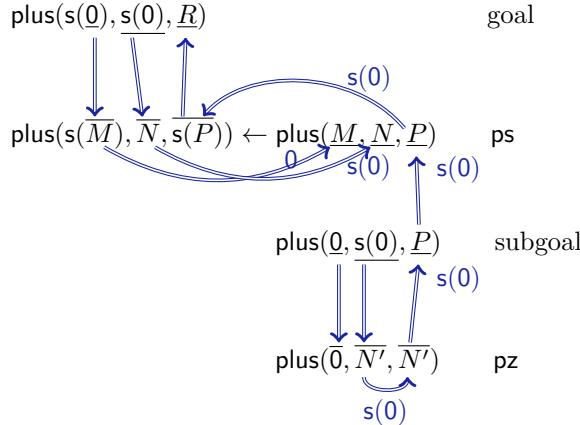
Without formally describing an algorithm yet, the intuitive idea is to match up corresponding subterms. If one of them is a variable, we set it to the other term. Here we set  $M = 0$ ,  $N = \text{s}(0)$ , and  $R = \text{s}(P)$ .  $P$  is arbitrary and remains a variable. Applying these equations to the body of the clause we obtain  $\text{plus}(0, \text{s}(0), P)$  which will be the subgoal with another logic variable,  $P$ .

In order to use the other clause for  $\text{plus}$  to solve this goal we have to solve  $\text{plus}(0, \text{s}(0), P) = \text{plus}(0, N, N)$  which sets  $N = \text{s}(0)$  and  $P = \text{s}(0)$ . The

---

<sup>7</sup>This section not covered in lecture

basic idea behind unification and the intuitive order how it works in this case is illustrated in the following diagram:



This process is called *unification*, and the equations for the variables we generate represent the *unifier*. In the above example the unifier unifying the given goal and the ps clause is

$$(0/M, s(0)/N, s(P)/R)$$

The concrete answer substitution  $(s(s(0))/R)$  will be found when the remaining computation terminates as in the diagram.

There are some subtle issues in unification. One is that the variables in the clause (which really are schematic variables in an inference rule) should be renamed to become fresh variables each time a clause is used so that the different instances of a rule are not confused with each other. This step is also called *standardize apart*, because it renames schematic variables to make them unique. Another issue is exemplified by the equation  $N = s(s(N))$  which does not have a solution: the right-hand side will have two more successors than the left-hand side so the two terms can never be equal. Unfortunately, Prolog does not properly account for this and treats such equations incorrectly by building a circular term (which is definitely not a part of the underlying logical foundation). This would come up if we pose the query  $\text{plus}(0, N, s(s(N)))$ : “Is there an  $n$  such that  $0 + n = n + 2$ .”

We discuss the reasons for Prolog’s behavior later in this course (which is related to efficiency), although we do not subscribe to it because it subverts the logical meaning of programs.

We will come back to a full discussion of unification at a later lecture. For the moment, this intuitive account of unification will suffice for our purposes.

## 10 Beyond Prolog

Since logic programming rests on an operational interpretation of logic, we can study various logics as well as properties of proof search in these logics in order to understand logic programming. In this way we can push the paradigm to its limits without departing too far from what makes it beautiful: its elegant logical foundation.

Ironically, even though logic programming derives from logic, the language we have considered so far (which is the basis of Prolog) does not require any logical connectives at all, just the mechanisms of judgments and inference rules. Extensions of it do lead to logical connectives, though.

## 11 Historical Notes

Logic programming and the Prolog language are credited to Alain Colmerauer and Robert Kowalski in the early 1970s. Colmerauer had been working on a specialized theorem prover for natural language processing, which eventually evolved to a general purpose language called Prolog (for *Programmation en Logique*) that embodies the operational reading of clauses formulated by Kowalski. Interesting accounts of the birth of logic programming can be found in papers by the Colmerauer and Roussel [1] and Kowalski [2].

We like Sterling and Shapiro's *The Art of Prolog* [4] as a good introductory textbook for those who already know how to program and we recommends O'Keefe's *The Craft of Prolog* as a second book for those aspiring to become real Prolog hackers. Both of these are somewhat dated and do not cover many modern developments, which are the focus of this course. We therefore do not use them as textbooks here.

## References

- [1] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *Conference on the History of Programming Languages (HOPL-II), Preprints*, pages 37–52, Cambridge, Massachusetts, April 1993.
- [2] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

- [3] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.
- [4] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 2nd edition edition, 1994.

# Lecture Notes on Prolog

15-317: Constructive Logic  
Frank Pfenning

Lecture 15  
October 24, 2017

## 1 Introduction

Prolog is the first and still standard backward-chaining logic programming language. While it is somewhat dated, it contains many interesting ideas and is appropriate for certain types of applications that naturally involve symbolic representations, backtracking, or unification. On the other, it has more than its typical share of pitfalls and problems, due to its dynamically typed nature, the prevalence of failure and backtracking, and the interactions between logical and extralogical predicates.

In this lecture we give a somewhat nonstandard introduction to Prolog by introducing a number of critical features using two examples: (1) basic computation on binary numbers in little endian representation, and (2) a proof checker based [Lecture 13](#) on certifying theorem provers.

## 2 Binary Numbers

Unary numbers, such as used in the Peano's axioms, are foundationally adequate, but not useful for practical computation due to the size of their representation. Instead, we use binary numbers. Representing them as *terms* in logic is straightforward, we just have to decide on the particulars. It turns out a so-called "little endian" representation where the least significant bits is the outermost constructor is most convenient. This is because when defining and operation on two numbers, say, the least significant bits

of both numbers line up correctly, and then we can recurse on the remainder of the numbers representing the higher bits.

$$\text{Binary numbers } M ::= \text{b0}(M) \mid \text{b1}(M) \mid e$$

where the (mathematical) translations between (mathematical) values and their representations are given by

$$\begin{array}{lll} \lceil 0 \rceil & = & e \\ \lceil 2n \rceil & = & \text{b0}(\lceil n \rceil) \text{ for } n > 0 \\ \lceil 2n + 1 \rceil & = & \text{b1}(\lceil n \rceil) \end{array} \quad \begin{array}{lll} \lfloor e \rfloor & = & 0 \\ \lfloor \text{b0}(M) \rfloor & = & 2 \lfloor M \rfloor \\ \lfloor \text{b1}(M) \rfloor & = & 2 \lfloor M \rfloor + 1 \end{array}$$

Now we can specify the successor relation  $\text{inc}(M, N)$  such that  $N$  represents the success of  $M$  by the following three rules:

$$\frac{}{\text{inc}(e, b1(e))} \text{inc}_e \quad \frac{}{\text{inc}(\text{b0}(M), \text{b1}(M))} \text{inc}_0 \quad \frac{\text{inc}(M, N)}{\text{inc}(\text{b1}(M), \text{b0}(N))} \text{inc}_1$$

In these rule, we use upper case variables for schematic variables in the rules, which is consistent with the Prolog syntax. We use lower case identifiers for predicates (inc, so far), constants (e), and function symbols (b0 and b1). In Prolog syntax, we write a rule

$$\frac{J_1 \dots J_n}{J}$$

as

$$J :- J_1, \dots, J_n.$$

which we read as " $J$  if  $J_1$  through  $J_n$ ". We call the rule a *clause*,  $J$  the *head of the clause* and  $J_1, \dots, J_n$  the *body of the clause*. If there are zero premises, the rules is simply written as ' $J$ '. Transcribing the rules then yields the following Prolog program. We call the predicate `inc_` (with a trailing underscore) to distinguish it from the predicate `inc` which fixes some of its problems.

```
inc_(e, b1(e)).  
inc_(b0(M), b1(M)).  
inc_(b1(M), b0(N)) :- inc_(M, N).
```

As we will see, there are some problems with this program. But first, let's fire up the Gnu Prolog interpreter to run this program on some inputs. The first line after the prompt `| ?-` is Prolog's notation for loading a program from a file, here `bin.pl`.

```
% gprolog
GNU Prolog 1.4.4 (64 bits)
Compiled Apr 23 2013, 17:26:17 with /opt/local/bin/gcc-apple-4.2
By Daniel Diaz
Copyright (C) 1999-2013 Daniel Diaz
| ?- ['bin.pl'].
...
(1 ms) yes
| ?-
```

At the prompt we can now type *queries* and have the interpreter simultaneously search for a proof and an instantiation of the free variables in the query. Once a solution has been found, the interpreter may give you the opportunity to search for other solutions, or simply return to the prompt if it can see no further solutions are possible. For example, we can increment 5 to get 6.

```
| ?- inc_(b1(b0(b1(e))),N).
N = b0(b1(b1(e)))
yes
| ?-
```

We can also give several goals conjunctively, and they will be solved in sequence. The following increments 5 three times to obtain 8.

```
| ?- inc_(b1(b0(b1(e))),N1), inc_(N1,N2), inc_(N2,N3).
N1 = b0(b1(b1(e)))
N2 = b1(b1(b1(e)))
N3 = b0(b0(b0(b1(e)))))

yes
| ?-
```

The fact that Prolog is dynamically typed, leads to some unexpected and meaningless answers:

```
| ?- inc_(b0(some_random_junk), N).
N = b1(some_random_junk)
```

This comes under the heading of “garbage-in, garbage-out”, but it is still disconcerting this would be claimed as true rather than meaningless.

Now let's try to run the predicate “in reverse” to calculate the predecessor of a binary number, in this case 6.

```
| ?- inc_(M, b0(b1(b1(e)))).  
M = b1(b0(b1(e))) ? ;  
no  
| ?-
```

Indeed, we obtain 5! Prolog asks if we would like to search for another solution, we type the semicolon verb ‘;’ and it confirms there are no further solutions. So far, things look good. Let's try the predecessor of 0, which should not exist.

```
| ?- inc_(M, e).  
no  
| ?-
```

Once again correct. Let's try the predecessor of 1:

```
| ?- inc_(M, b1(e)).  
M = e ? ;  
M = b0(e) ? ;  
no  
| ?-
```

Here we have a surprise: we get two answers! The second one also morally represents the number  $2 \cdot 0 = 0$ , but it is not in standard form since it has a leading bit b0. The problem is that both the first and second rules apply to this query

```
inc_(e, b1(e)).  
inc_(b0(M), b1(M)).  
inc_(b1(M), b0(N)) :- inc_(M, N).
```

Returning an answer not in standard form is a problem only if we want to always maintain standard form (which seems like a good idea). But even if we do not, the fact that the innocuous looking predicate returns a second answer upon backtracking will almost certainly lead to unintended consequences wherever this predicate is used.

If we want to run this predicate with the mode `inc_(-std, +std)` then we need to prevent the second solution by distinguishing the cases for  $M$  in the middle clause.

```
inc(e, b1(e)) .  
inc(b0(b0(M)), b1(b0(M))) .  
inc(b0(b1(M)), b1(b1(M))) .  
inc(b1(M), b0(N)) :- inc(M, N) .
```

Now  $b0(e)$  is ruled out, and the problem disappears:

```
| ?- inc(M, b1(e)) .
```

```
M = e ? ;
```

```
no  
| ?-
```

Here is one way to define what it means to be in standard form

```
std(e) .  
% no case for std(b0(e))  
std(b0(b0(N))) :- std(b0(N)) .  
std(b0(b1(N))) :- std(b1(N)) .  
std(b1(N)) :- std(N) .
```

We will see another way in the next lecture.

### 3 Checking Proof Terms

We now move on to another example, which introduces a number of other features of Prolog: proof checking. The rules were introduced in [Lecture 13](#) and we only summarize the fragment with implication and conjunction here.

$$\begin{array}{ll} \text{Checkable terms} & M, N ::= (M, N) \mid (\text{fn } u \Rightarrow M) \mid R \\ \text{Synthesizing terms} & R ::= u \mid \text{fst } R \mid \text{snd } R \mid R M \\ \text{Ordered contexts} & \Omega ::= . \mid (u : A \downarrow) \cdot \Omega \end{array}$$

We use the ordered context so we can check terms such as

$$\vdash (\text{fn } x \Rightarrow \text{fn } x \Rightarrow x) : a \supset (b \supset b)$$

where  $x$  refers to the innermost binding of  $x$ , but not to any other ones:

$$\not\vdash (\text{fn } x \Rightarrow \text{fn } x \Rightarrow x) : a \supset (b \supset a)$$

We have the following rules.

$$\begin{array}{c} \frac{\Omega \vdash M : A \uparrow \quad \Omega \vdash N : B \uparrow}{\Omega \vdash (M, N) : A \wedge B \uparrow} \wedge I \quad \frac{(u : A \downarrow) \cdot \Omega \vdash M : B \uparrow}{\Omega \vdash (\text{fn } u \Rightarrow M) : A \supset B \uparrow} \supset I^u \\ \\ \frac{\Omega \vdash R : A \downarrow}{\Omega \vdash R : A \uparrow} \downarrow \uparrow \\ \\ \frac{}{(u : A \downarrow) \cdot \Omega \vdash u : A \downarrow} \text{var}_= \quad \frac{w \neq u \quad \Omega \vdash u : A \downarrow}{(w : B \downarrow) \cdot \Omega \vdash u : A \downarrow} \text{var}_\neq \\ \\ \frac{\Omega \vdash R : A \wedge B \downarrow}{\Omega \vdash \text{fst } R : A \downarrow} \wedge E_1 \quad \frac{\Omega \vdash R : A \wedge B \downarrow}{\Omega \vdash \text{snd } R : B \downarrow} \wedge E_2 \\ \\ \frac{\Omega \vdash R : A \supset B \downarrow \quad \Omega \vdash M : A \uparrow}{\Omega \vdash RM : B \downarrow} \supset E \end{array}$$

We have the intuition that these rules describe an algorithm, but now the (almost) really do describe a program, in Prolog! We have two predicates, one (`check/3`, which means `check` with 3 arguments) in which  $\Omega$ ,  $M$ , and  $A$  must all be given and it succeeds or fails, and `synth/3` in which  $\Omega$  and  $R$  are given and it synthesizes  $A$  or fails.

```
check (+O, +M, +A)
synth (+O, +R, -A)
```

The first rule is easy to translate, using the constructors `pair(M, N)` for  $(M, N)$  and `and(A, B)` for  $A \wedge B$ :

```
check (O, pair (M, N), and (A, B)) :-  
    check (O, M, A),  
    check (O, N, B).
```

In the second rule,  $\supset I$ , we have to add a variable and its type to the front of the list  $\Omega$ . The syntax for lists in Prolog allows several different notations. We have the empty list `[]`, and we have a constructor which takes an element  $x$  and adds it to the front of the list  $xs$  with `'.'(x, xs)`. Here, use of “dot” requires single quotes so it is not confused with the end-of-clause period. An alternative notation for `'.'`( $x, xs$ ) is `[x | xs]`, which is commonly used. Finally we can also write out lists as with `[1, 2, 3, 4]`. All of the following denote the same list:

```
[1,2,3,4]    [1|[2,3,4]]    [1,2|[3,4]]
'.'(1,[2,3,4])  ').'(1,'.'(2,'.'(3,'.'(4,[]))))
[1|[2|[3|[4|[]]]]]
```

We use the most common syntax to add `tp(X, A)` to the front of the context  $\Omega$ , where `tp/2` is a new Prolog term constructor.

```
check(O, fun(X,M), imp(A,B)) :-  
    check([tp(X,A)|O], M, B).
```

If neither of these two clauses match, we could be looking at a synthesizable term  $R$ , so we should try to synthesize a type for it and compare it to the given one.

```
check(O, R, A) :- synth(O, R, B), A = B.
```

Here we use the built-in equality predicate to *unify A and B*. In this case, for a mode-correct query,  $A$  is given and input to `check` and  $B$  will be returned by `synth`, so the comparison is just an equality test, not full unification.

The synthesis judgment is again straightforward for pairs.

```
synth(O, fst(M), A) :- synth(O, M, and(A,B)).  
synth(O, snd(M), B) :- synth(O, M, and(A,B)).
```

We can see this is mode-correct for `synth(+, +, -)`. In the head of the clause, we know  $O$  and `fst(M)` and therefore  $M$ . Now we can appeal to the hypothesis that `and(A, B)` will be known if the subgoal succeeds, which means  $A$  is known, which is what we needed to show.

In the case of application `app(R, M)` we just need to be careful to solve the two subgoals in the right order.

```
synth(O, app(R,M), B) :-  
    synth(O, R, imp(A,B)),  
    check(O, M, A).
```

If we had switched them, as in

```
synth(O, app(R,M), B) :-  
    check(O, M, A),          % bug here!  
    synth(O, R, imp(A,B)).
```

it would not be mode correct: when we call `check(O, M, A)` we do not yet know `A`, which is required for `check/3`.

Finally we have two rules for variables, where we either find `X` at the head of the list, or look for it in the tail.

```
% warning: these have a bug!  
synth([tp(X,A)|O], X, A).  
synth([tp(Y,B)|O], X, A) :- synth(O, X, A).
```

Let's run some examples with this code to test it. The first one checks the identity function, then we check the first and second projections. All succeed and fail as expected.

```
| ?- check([], fun(x,x), imp(a,a)).  
  
true ? ;  
  
no  
| ?- check([], fun(x,fun(y,x)), imp(a,imp(b,a))).  
  
true ? ;  
  
no  
| ?- check([], fun(x,fun(y,y)), imp(a,imp(b,b))).  
  
true ? ;  
  
no  
| ?-
```

However, there is a bug in the program. Consider

```
| ?- check([], fun(x,fun(x,x)), imp(a,imp(b,b))).  
  
true ? ;
```

```

no
| ?- check([], fun(x, fun(x, x)), imp(a, imp(b, a))) .

true ? ;

no
| ?-

```

The first query succeeds as expected, because  $x$  should refer to its innermost enclosing binder. The second query shows that we incorrectly allow  $x$  to also refer to the outer binder!

```

% warning: these have a bug!
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- synth(O, X, A).

```

If we trace the execution we can see the problem

```

| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- check([], fun(x, fun(x, x)), imp(a, imp(b, a))) .
   1   1  Call: check([], fun(x, fun(x, x)), imp(a, imp(b, a))) ?
   2   2  Call: check([tp(x,a)], fun(x,x), imp(b,a)) ?
   3   3  Call: check([tp(x,b), tp(x,a)], x, a) ?
   4   4  Call: synth([tp(x,b), tp(x,a)], x, _416) ?
   4   4  Exit: synth([tp(x,b), tp(x,a)], x, b) ?
   4   4  Redo: synth([tp(x,b), tp(x,a)], x, b) ?
   5   5  Call: synth([tp(x,a)], x, _441) ?
   5   5  Exit: synth([tp(x,a)], x, a) ?
   4   4  Exit: synth([tp(x,b), tp(x,a)], x, a) ?
   3   3  Exit: check([tp(x,b), tp(x,a)], x, a) ?
   2   2  Exit: check([tp(x,a)], fun(x,x), imp(b,a)) ?
   1   1  Exit: check([], fun(x, fun(x, x)), imp(a, imp(b, a))) ?

true ?

```

In the 5th line of the trace, we exit the query  $\text{synth}([\text{tp}(x,b), \text{tp}(x,a)], x, \_416)$  ? with  $\_416 = b$ . However, this fails to unify with the  $a$  we are supposed to check against in line 3! So Prolog backtracks (see Redo) and synthesizes another type from the remainder of the context  $O$ , namely  $a$ . This now works and the query incorrectly succeeds.

The fix requires that we only continue to look through the context if the variable  $x$  we are trying to find is *different* from the variable at the head of the list:

```
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- Y \= X, synth(O, X, A).
```

The new goal  $Y \neq X$  stands for “ $Y$  and  $X$  are not unifiable”, although here both  $Y$  and  $X$  will be known and it just comes down to checking equality between two terms.

Here is the summary of the repaired program

```
% check(+O, +M, +A)
% synth(+O, +R, -A)

check(O, pair(M,N), and(A,B)) :-
    check(O, M, A),
    check(O, N, B).
check(O, fun(X,M), imp(A,B)) :-
    check([tp(X,A)|O], M, B).
check(O, R, A) :- synth(O, R, B), A = B.

synth(O, fst(M), A) :- synth(O, M, and(A,B)).
synth(O, snd(M), B) :- synth(O, M, and(A,B)).
synth(O, app(R,M), B) :-
    synth(O, R, imp(A,B)),
    check(O, M, A).
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- Y \= X, synth(O, X, A).
```

Let’s run a few more queries. First the uncurrying function, which proves  $(A \supset (B \supset C)) \supset ((A \wedge B) \supset C)$  and then uncurrying, which goes in the opposite direction.

```
| ?- check([], fun(f, fun(p, app(app(f, fst(p)), snd(p)))), imp(imp(a, imp(b, c)), imp(and(a, b), c))).

true ? ;

no
| ?- check([], fun(f, fun(x, fun(y, app(f, pair(x, y))))), imp(imp(and(a, b), c), imp(a, imp(b, c)))).

true ? ;

(1 ms) no
```

## 4 Unification

This has worked out extremely well so far, so now we are getting greedy. What about running `check/3` in a mode such as `check(+O, +M, -A)` which amounts to type inference? Let's try!

```
| ?- check([], fun(x,x), A).
A = imp(B,B) ? ;
no
```

The interpreter finds one solution, namely  $B \supset B$  for any  $B$ . In type inference we can this *the most general type* since any other type is an instance of it. Let's try something more complicated, such as  $\text{fn } f \Rightarrow \text{fn } x \Rightarrow \text{fn } y \Rightarrow f(x,y)$ :

```
| ?- check([], fun(f,fun(x,fun(y,app(f,pair(x,y))))), A).
A = imp(imp(and(B,C),D),imp(B,imp(C,D))) ? ;
no
```

Again, the interpreter finds a single most general solution, namely  $((B \wedge C) \supset D) \supset (B \supset (C \supset D))$  for any propositions (types)  $B, C$ , and  $D$ . During the search for a proof of the goal, the interpreter accumulates constraints on  $A$ . These are then solved by a process called *unification*, which fails if there are no solutions or simplifies the constraints to a minimal form where it is easy to read off the solution in the form of a substitution.

Unfortunately, Prolog's algorithm for unification is *unsound* for reasons of efficiency. This is really inexcusable, especially since the overhead together with other optimizations is not very high, but we now have to live with that. To see the problem consider the term  $\text{fn } x \Rightarrow x\ x$ . There should not be a type for this term, because after a couple of steps we are in the situation (writing unknown types as greek letters):

$$\frac{\frac{x}{(x : \alpha \downarrow) \vdash x : \alpha \downarrow} x \quad \frac{x : \alpha \downarrow \vdash x : \alpha \downarrow \quad \alpha = \beta}{x : \alpha \downarrow \vdash x : \beta \uparrow} \downarrow \uparrow \quad \alpha = \beta \supset \gamma}{x : \alpha \downarrow \vdash x\ x : \gamma \downarrow} \supset E$$

The reason that type inference fails is that there is no solution to the equations  $\alpha = \beta, \alpha = \beta \supset \gamma$  because  $\beta = \beta \supset \gamma$  has no solution.

Surprisingly, Prolog fails to notice that. Or, more precisely, it builds a cyclic term to solve this equation.

```
check([], fun(x, app(x, x)), A).
cannot display cyclic term for A ? ;
no
```

Essentially, when it processes the equation  $\beta = B$  for some type  $B$  it just sets  $\beta$  to be equal to  $B$  without checking if this would introduce a cyclic term. In Prolog terminology we say that it *omits the occurs check* which would verify that  $B$  does not contain  $\beta$ .

Because of this shortcoming, Prolog has an explicit predicate `unify_with_occurs_check/2` that makes sure the two arguments are unified properly, so that the problem  $\beta = \beta \supset \gamma$  fails. Wherever in the program a variable is repeated in the head of a clause, or we call on unification, we should call on unification with the occurs check instead. Fortunately, we only have to consider three lines. First

```
check(O, R, A) :- synth(O, R, B), A = B.
```

Now that we have a more general mode (allowing the third argument to be partially instantiated, but contain free variables), we can rewrite it as

```
check(O, R, A) :- synth(O, R, A).
```

Second, we look at the two lines where variables are considered

```
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- Y \= X, synth(O, X, A).
```

In the first line, the two occurrences of  $A$  are unified, which could be unsound so we need to appeal to `unify_with_occurs_check/2` instead. There is no such problem in the second line, because  $B$  and  $A$  are unrelated. Here is the complete revised program:

```

check(O, pair(M,N), and(A,B)) :-  

    check(O, M, A),  

    check(O, N, B).  

check(O, fun(X,M), imp(A,B)) :-  

    check([tp(X,A)|O], M, B).  

check(O, R, A) :- synth(O, R, A).  
  

synth(O, fst(M), A) :- synth(O, M, and(A,B)).  

synth(O, snd(M), B) :- synth(O, M, and(A,B)).  

synth(O, app(R,M), B) :-  

    synth(O, R, imp(A,B)),  

    check(O, M, A).  

synth([tp(X,B)|O], X, A) :- unify_with_occurs_check(B, A).  

synth([tp(Y,B)|O], X, A) :- Y \= X, synth(O, X, A).

```

Now we find that self-application is not typable, as expected.

```
| ?- check([], fun(x, app(x,x)), A).
```

no

while the other, positive examples continue to work as expected.

In a future lecture we will show the specifics about how unification in Prolog as well as sound unification works.

This particular example is a remarkable compact implementation of full type inference for a small language. Intrinsic pattern matching is critical, as it is the built-in notion of logic variable and unification. Backtracking does not particularly come into play here, but it will if you implement the G4ip decision procedure based on Dyckhoff's contraction-free sequent calculus.

An attempt to use this proof checker as a theorem prover predictably fails:

```
| ?- check([], M, imp(a,a)).
```

```
Fatal Error: local stack overflow ...
```

# Lecture Notes on Types as Predicates

15-317: Constructive Logic  
Frank Pfenning

Lecture 16  
October 26, 2017

## 1 Introduction

One of the significant problems in using Prolog is the lack of static typing. Prolog inherited this feature from *predicate calculus*, where it roots lie. In the foundational study of propositions and quantification, types are often omitted because it is said they can already be expressed. For example, instead of saying  $\forall x:\text{nat}. A(x)$  we can say  $\forall x. \text{nat}(x) \supset A(x)$  if we have a *predicate* nat that expresses the *type* nat. Similarly, we can express  $\exists x:\text{nat}. A(x)$  as  $\exists x. \text{nat}(x) \wedge A(x)$ . Predicates that provide an extensional representation of types are not difficult to come by. For example, we can define (and have defined) the natural numbers with two constructors z and s and the rules

$$\frac{}{\text{nat}(z) \text{ true}} \text{ nat}_z \quad \frac{\text{nat}(N) \text{ true}}{\text{nat}(s(N)) \text{ true}} \text{ nat}_s$$

Foundationally, this approach may have some merit, but it also has some problems. One is that propositions such as  $\forall x:\text{nat}. \text{append}(x, \text{nil}, x)$  which are *meaningless* become either true or false when written in an untyped way:  $\forall x. \text{nat}(x) \supset \text{append}(x, \text{nil}, x)$ . In a language like Prolog this has dire consequences because we compute with intuitively meaningless propositions and bogus proofs, leading to unexpected behavior. A second problem is that the untyped approach does not extend well to higher-order logic, where we want to quantify over propositions and not just data. In fact, several times in history well-regarded researchers such as Frege or Church have attempted to avoid the organizing principles of types, leading to inconsistent logics.

In this lecture we ask explore the question if we may still be able to use the idea of defining types via (unary) predicates and obtain something we can statically check and that executes efficiently at the same time. The answer is “yes”, and the lessons learned from this has also had some impact on functional programming in the guise of *refinement types* [FP91, DP03, Dav97].

There have been multiple approaches to types in the logic programming community (see [Pfe92] for various articles and technical realizations). We will not go into a specific decidable language of types, although much of what we show in this lecture applies to several systems that are different in the technical details.

## 2 Modes and Types

Let’s reconsider something simple like addition on unary natural numbers.

$$\begin{array}{c} \frac{}{\text{nat}(\text{z})} \text{ nat}_z \quad \frac{\text{nat}(N)}{\text{nat}(\text{s}(N))} \text{ nat}_s \\ \frac{}{\text{plus}(\text{z}, N, N)} \text{ pz} \quad \frac{\text{plus}(M, N, P)}{\text{plus}(\text{s}(M), N, \text{s}(P))} \text{ ps} \end{array}$$

Now we want to show the combined mode and type specification:

$$\text{plus}(\text{+nat}, \text{+nat}, \text{-nat})$$

which we interpret as follows: if proof search is initiated with a goal  $\text{plus}(m, n, P)$  where  $\text{nat}(m)$  and  $\text{nat}(n)$  and *succeeds*, then  $P = p$  with  $\text{nat}(p)$ .

Rigorously, we would have to prove this by induction over the structure of computation (that is, proof search). In the absence of such an operational semantics, we prove it by induction over the structure of the rules. Assume we are searching for a proof of  $\text{plus}(m, n, P)$  for a variable  $P$  and terms  $m$  and  $n$  with  $\text{nat}(m)$  and  $\text{nat}(n)$ .

**Case:** Rule pz. We know  $\text{nat}(\text{z})$  (which adds no new information) and  $\text{nat}(n)$ . Applying the rule will succeed and instantiate  $P = n$  and so  $\text{nat}(P)$ .

**Case:** Rule ps. We know  $m = \text{s}(m')$  and  $\text{nat}(\text{s}(m'))$  and also  $\text{nat}(n)$ . From the first fact, by inversion (only rule  $\text{nat}_s$  could be used to prove this)

we obtain  $\text{nat}(m')$ . Now we can appeal to the induction hypothesis: if the subgoal  $\text{plus}(m', n, P')$  succeeds, the  $P' = p'$  and  $\text{nat}(p')$ . Then  $\text{nat}(\text{s}(p'))$  by rule  $\text{nat}_s$ .

So far, there is not much new or interesting in this when compared to types as we know them from functional languages. But we can define new and interesting types as predicates and reason about them in the same style. For example, we can distinguish the even and odd numbers and reason about the properties of addition.

$$\frac{}{\text{even}(\text{z})} \text{ ev}_z \quad \frac{\text{odd}(N)}{\text{even}(\text{s}(N))} \text{ ev}_s \quad \frac{\text{even}(N)}{\text{odd}(\text{s}(N))} \text{ od}_s$$

Let's try to check that adding two even numbers results in an even number.

$$\frac{}{\text{plus}(\text{z}, N, N)} \text{ pz} \quad \frac{\text{plus}(M, N, P)}{\text{plus}(\text{s}(M), N, \text{s}(P))} \text{ ps}$$

$$\text{plus}(\text{+even}, \text{+even}, \text{-even})$$

**Case:** Rule pz.

$$\begin{array}{ll} \text{even}(\text{z}) & \text{mode +even of first arg.} \\ \text{even}(N) & \text{mode +even of second arg.} \\ \text{even}(N) & \text{previous line} \end{array}$$

**Case:** Rule ps.

$$\begin{array}{ll} \text{even}(\text{s}(M)) & \text{mode +even of first arg.} \\ \text{odd}(M) & \text{by inversion from previous line} \\ \text{even}(N) & \text{mode +even of second arg.} \end{array}$$

At this point we are stuck because we cannot apply the induction hypothesis, only knowing that  $M$  is odd.

So we need to generalize our declaration to

- (i)  $\text{plus}(\text{+even}, \text{+even}, \text{-even})$
- (ii)  $\text{plus}(\text{+odd}, \text{+even}, \text{-odd})$

and restart our proof.

**Case (i):** Rule pz.

even(z)	mode +even of first arg.
even( $N$ )	mode +even of second arg.
even( $N$ )	previous line

**Case (i):** Rule ps.

even(s( $M$ ))	mode +even of first arg.
odd( $M$ )	by inversion from previous line
even( $N$ )	mode +even of second arg.
odd( $P$ )	by i.h.(ii)
even(s( $P$ ))	by rule ev <sub>s</sub>

**Case (ii):** Rule pz.

odd(z)	mode +odd of first arg.
Contradiction	by inversion (no rule concluding odd(z))

**Case (ii):** Rule ps.

odd(s( $M$ ))	mode +odd of first arg.
even( $M$ )	by inversion from previous line
even( $N$ )	mode +even of second arg.
even( $P$ )	by i.h.(i)
odd(s( $P$ ))	by rule od <sub>s</sub>

Note there that the case pz in the proof of (ii) is impossible: the rule pz cannot apply if the first argument of plus is odd. From the contradiction in this case we can infer anything, in particular that the third argument will be odd if the search succeeds, which it never will.

We see two differences here already to a system of types for languages such as ML: we have a natural notion of multiple related types (such as even and odd numbers, as well as arbitrary natural numbers), and a given predicate such as plus may have multiple types, all of them necessary for type-checking purposes.

### 3 Subtyping

Types defined as predicates come with a natural notion of subtyping. For two predicates  $s$  and  $t$  we write  $s \leq t$  if  $\forall x. s(x) \supseteq t(x)$ , that is, every element satisfying  $s$  also satisfies  $t$ .

To appreciate the need for subtyping, we consider once again binary numbers and numbers in standard form (no leading zero's). We defined this slightly differently from last lecture by stipulating that in a term  $b0(N)$ ,  $N$  must be positive. This enforces that it cannot be  $e$ , which represents zero and is therefore not positive.

$$\begin{array}{c} \text{---} \quad \text{std}_e \\ \text{std}(e) \end{array} \quad \begin{array}{c} \frac{\text{pos}(N)}{\text{std}(b0(N))} \quad \text{std}_0 \\ \text{std}(b1(N)) \quad \text{std}_1 \end{array} \quad \begin{array}{c} \frac{\text{std}(N)}{\text{std}(b1(N))} \quad \text{std}_1 \\ \text{std}(b0(N)) \quad \text{pos}_1 \end{array}$$

$$\begin{array}{c} \text{no rule pos}_e \\ \frac{\text{pos}(N)}{\text{pos}(b0(N))} \quad \text{pos}_0 \end{array}$$

We now recall the increment predicate and try to verify that, if given a standard number it will construct a positive one.

$$\begin{array}{c} \text{---} \quad \text{inc}_e \\ \text{inc}(e, b1(e)) \end{array} \quad \begin{array}{c} \text{---} \quad \text{inc}_0 \\ \text{inc}(b0(M), b1(M)) \end{array} \quad \begin{array}{c} \frac{\text{inc}(M, N)}{\text{inc}(b1(M), b0(N))} \quad \text{inc}_1 \\ \text{inc}(+std, -pos) \end{array}$$

**Case:** Rule  $\text{inc}_e$ .

$$\text{pos}(b1(e)) \quad \text{by rules pos}_1 \text{ and std}_e$$

**Case:** Rule  $\text{inc}_0$ .

$$\begin{array}{ll} \text{std}(b0(M)) & \text{first arg.} \\ \text{pos}(M) & \text{by inversion} \\ \text{std}(M) & \text{by pos} \leq \text{std}, \text{ see below} \\ \text{pos}(b1(M)) & \text{by rule pos}_1 \end{array}$$

**Case:** Rule  $\text{inc}_1$ .

$$\begin{array}{ll} \text{std}(b1(M)) & \text{first arg.} \\ \text{std}(M) & \text{by inversion} \\ \text{pos}(N) & \text{by i.h.} \\ \text{pos}(b0(N)) & \text{by rule pos}_0 \end{array}$$

At this point the proof is complete, if we can show that  $\text{pos} \leq \text{std}$ . This is now a property that no longer requires appeal to the definition of  $\text{inc}$ ; it is just a property of the two types. We can proceed by induction (actually, just a proof by cases is required) on the definition of  $\text{pos}$ .

**Case:** Rule  $\text{pos}_0$ .

$$\begin{array}{ll} \text{pos}(N) & \text{premise} \\ \text{std}(\text{b}0(N)) & \text{by rule std}_0 \end{array}$$

**Case:** Rule  $\text{pos}_1$ .

$$\begin{array}{ll} \text{std}(N) & \text{premise} \\ \text{std}(\text{b}0(N)) & \text{by rule std}_1 \end{array}$$

Next we see how this kind of static type checking (phrased here as theorem proving) can help uncover errors. For example, we may want to check that

$$\text{inc}(-\text{std}, +\text{std})$$

**Case:** Rule  $\text{inc}_e$ . Then  $\text{std}(\text{e})$ .

**Case:** Rule  $\text{inc}_0$ .

$$\begin{array}{ll} \text{std}(\text{b}1(N)) & \text{second arg. of inc} \\ \text{std}(N) & \text{by inversion} \\ \text{Need: pos}(N) & \text{not true in general!} \\ \text{std}(\text{b}0(N)) & \text{by rule std}_0 \end{array}$$

There is no way to fix the missing step in the second case (we didn't even get around to the third case).  $\text{std}(N)$  does not imply  $\text{pos}(N)$ , with  $N = \text{e}$  as a counterexample. Indeed, one solution for

$$\text{?- inc}(M, \text{b}1(\text{e}))$$

is  $M = \text{b}0(\text{e})$  which is *not* in standard form.

At this point we might consider some other properties. Let's define some new types, such as  $\text{zero}(N)$  and  $\text{empty}(N)$ :

$$\frac{}{\text{zero}(\text{e})} \quad \text{zero}_e \quad \text{no rule for empty}(N)$$

Now we can show, with type checking that a query  $\text{inc}(M, e)$  cannot succeed. The type we ascribe is

$$\text{inc}(-\text{empty}, +\text{zero})$$

which expresses that if a query  $\text{inc}(M, n)$  with  $\text{zero}(n)$  succeeds with  $M = m$ , then  $\text{empty}(m)$ . Since there is no such  $m$ , this means *if inc has the given type* then decrementing zero can not succeed. This means it either doesn't terminate or it fails after a finite number of steps.

Now to the type checking:

**Case:** Rule  $\text{inc}_e$ .

$$\begin{array}{ll} \text{zero(b1(e))} & \text{second argument} \\ \text{Contradiction} & \text{by inversion (no rule concludes zero(b1(e)))} \end{array}$$

**Case:** Rule  $\text{inc}_0$ .

$$\begin{array}{ll} \text{zero(b1(M))} & \text{second argument} \\ \text{Contradiction} & \text{by inversion} \end{array}$$

**Case:** Rule  $\text{inc}_1$ .

$$\begin{array}{ll} \text{zero(b0(M))} & \text{second argument} \\ \text{Contradiction} & \text{by inversion} \end{array}$$

All cases are impossible, so the type  $\text{inc}(-\text{empty}, +\text{zero})$  is correct.

As a last example we revisited the even/odd distinction, now on binary numbers. We could just look at the least significant bit, but we arrange that  $\text{even} \leq \text{std}$  and  $\text{odd} \leq \text{pos}$  to ease working with these types.

$$\frac{\text{pos}(N)}{\text{even}(\text{b0}(N))} \text{ ev}_0 \quad \frac{\text{std}(N)}{\text{odd}(\text{b1}(N))} \text{ od}_1$$

We leave it to the reader to now verify that

$$\begin{array}{l} \text{inc}(+\text{even}, -\text{odd}) \\ \text{inc}(+\text{odd}, -\text{even}) \end{array}$$

## 4 Refinement types for functional languages

The idea that we have more precise types than just `nat` (like `even` and `odd`) or binary numbers (like `std`, `pos`, `zero`, `empty`) could be a priori useful for functional languages as well.

The main complication is that we also need to include *intersection types* [CDCV81, Rey91] to make this work. We retain the usual data types, but we add *data sort* declarations that declare *refinements* [FP91]. The examples in this section and many more can be found in a converservative extension of Standard ML with datasort refinements called Cidre [Dav97], available on GitHub<sup>1</sup>.

For example, we can define `even` and `odd` unary numbers as follows:

```
datatype nat = z | s of nat

datasort even = z | s of odd
and odd = s of even
```

But now if we have a simple function such as

```
fun succ x = s(x)
```

we find

```
succ : (nat -> nat) & (even -> odd) & (odd -> even)
```

so we need to be able to ascribe multiple types to a function or expression. This is what the intersection type operator  $A \sqcap B$  achieves, which we write as `A & B` in concrete syntax. Sometimes, several types are needed. For example

```
fun twice f x = f (f x)
```

then we should be able to show (among many other types)

```
twice : ((nat -> nat) -> nat -> nat)
& (((even -> odd) & (odd -> even)) -> (even -> even))
& (((even -> odd) & (odd -> even)) -> (odd -> odd))
```

The resulting system has some remarkable properties, such as decidability of type inference, bidirectional type checking, and conservative extension over ML. It is implemented in the Cidre front end, which accepts

---

<sup>1</sup><https://github.com/rowandavies/sml-cidre>

the full syntax of Standard ML and uses stylized comments to assign refinement types that are then checked.

You probably have seen one example where this might have been helpful. On the fragment of propositions with implication and conjunction only, we defined proof terms in Assignment 6 via the following data type

```
datatype term = Fun of var * term
              | Pair of term * term
              | Var of var
              | App of term * term
              | Fst of term
              | Snd of term
```

In a way, this was a compromise, since we distinguished, in the problem statement and the algorithm, between checkable and synthesizing terms. The corresponding data type declaration would be something like

```
datatype cterm = Fun of var * cterm
              | Pair of cterm * cterm
              | Syn of sterm
and sterm = Var of var
           | App of sterm * cterm
           | Fst of sterm
           | Snd of sterm
```

but there are two drawbacks: (1) we need to make the transition from synthesizing to checkable terms explicit (see `Syn` constructor), which complicates practical examples, and (2) now *everywhere* that terms are used, even in places where the distinction would be insignificant, we have to be cognizant and specific about whether we are working with checkable or synthesizing terms. With refinement types, we would first declare the type of terms, and then think of checkable and synthesizing terms as refinements.

```
datasort cterm = Fun of var * cterm
                | Pair of cterm * cterm
                | sterm
and sterm = Var of var
           | App of sterm * cterm
           | Fst of sterm
           | Snd of sterm
```

We can now freely use either `term` (where it doesn't matter) or `cterm` or `sterm` where the distinction is significant.

## References

- [CDCV81] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.
- [Dav97] Rowan Davies. A practical refinement-type checker for Standard ML. In Michael Johnson, editor, *Algebraic Methodology and Software Technology Sixth International Conference (AMAST'97)*, pages 565–566, Sydney, Australia, December 1997. Springer-Verlag LNCS 1349.
- [DP03] Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In A.D. Gordon, editor, *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03)*, pages 250–266, Warsaw, Poland, April 2003. Springer-Verlag LNCS 2620.
- [FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.
- [Pfe92] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.
- [Rey91] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Berlin, 1991. Springer-Verlag.

# Lecture Notes on Chaining

15-317: Constructive Logic  
Frank Pfenning

Lecture 17  
October 31, 2017

## 1 Introduction

There is a pleasing simplicity of the computation as proof search paradigm when based on inference rules. But how do we now implement it? One way would be to capture the rules as propositions so we can represent a program as a collection of propositions, which become the antecedents of a sequent. The query will then be the succedent, and we can hopefully use our usual tools of logical reasoning to obtain an operational semantics.

But what is the connection between rules and propositions, and can we reformulate the bottom-up search strategy using inference rules instead with antecedent propositions? This is the subject of today's lecture.

## 2 Rules as Propositions

Let's review a set of rules for increment in binary (omitting some fine points on standard representations of numbers).

$$\frac{}{\text{inc}(e, \text{b1}(e))} \text{ inc}_e \quad \frac{}{\text{inc}(\text{b0}(M), \text{b1}(M))} \text{ inc}_0 \quad \frac{\text{inc}(M, N)}{\text{inc}(\text{b1}(M), \text{b0}(N))} \text{ inc}_1$$

The first one  $\text{inc}_e$  is trivial to express as just an atomic proposition; the second ( $\text{inc}_0$ ) requires a simple quantifier. The third one has a premise, in which case we can rewrite the inference rule as an implication. Let's write

the resulting collection of antecedents as  $\Gamma_{\text{inc}}$ .

$$\begin{aligned}\Gamma_{\text{inc}} = & \text{ inc}(e, b1(e)), \\ & \forall m. \text{ inc}(b0(m), b1(m)), \\ & \forall m. \forall n. \text{ inc}(m, n) \supset \text{ inc}(b1(m), b0(n))\end{aligned}$$

We don't yet have the tools to see in which way this translation is correct, but at least in an intuitive sense it should look plausible.

Now, solving a logic programming query such as  $\text{inc}(b1(e), b1(b0(e)))$  will be modeled as

$$\Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(b1(e), b1(b0(e)))$$

where " $\xrightarrow{f}$ " demands a sequent calculus we have yet to design. Here we use  $f$  to suggest *focusing*, although in this lecture we will just introduce *chaining*. Focusing, eventually, will turn out to be *inversion + chaining*. The particular form of chaining from this lecture is called *backward chaining*; in the next lecture we also discuss forward chaining.

### 3 Chaining

We notice first that all propositions in  $\Gamma_{\text{inc}}$  are *noninvertible*: neither atoms, nor universal quantification, nor implication have invertible rules as antecedents. So the inversion strategy will not be useful. The second thing we notice is that *if we still had rules*, we would know immediately that only the  $\text{inc}_1$ , represented as the third antecedent, will actually help in the example at the end of the previous section. That is, we need to "look beyond" the quantifiers and to the right of the implication until we find an atomic formula and then decide if it would match our goal.

To formalize this, we *focus* on a particular proposition among the antecedents and then continue to apply rules *only to the proposition in focus* until we can determine whether it matches the succedent. We write  $[A]$  for the particular proposition  $A$  that is in focus. There can be at most one of those in a sequent.

The first rule picks a proposition from  $\Gamma$  (which, in our case, would be  $\Gamma_{\text{inc}}$ ) and puts it into focus. For now, the succedent will always be an atom, so we wrote  $P$ .

$$\frac{A \in \Gamma \quad \Gamma, [A] \xrightarrow{f} P}{\Gamma \xrightarrow{f} P} \text{ focusL}$$

The second rule instantiates a universal quantifier. Of course, at this point we cannot know which term we might want to choose, so we instantiate it with a variable  $X$ , to be determined later.

$$\frac{\Gamma, [A(X)] \xrightarrow{f} P}{\Gamma, [\forall x. A(x)] \xrightarrow{f} P} \forall L$$

When we reach an implication  $B \supset A$ , we would like to “look past” it at  $A$ , but we also have to remember that we still have to prove  $B$  in case this  $A$  does match the succedent.

$$\frac{\Gamma, [A] \xrightarrow{f} P \quad \Gamma \xrightarrow{f} [B]}{\Gamma, [B \supset A] \xrightarrow{f} P} \supset L$$

Two remarks: (1) we put  $\Gamma, [A] \xrightarrow{f} P$  as the first premise (reversing the order we have used so far for implication) because we would like to know if it matches the succedent before we solve  $B$ , and (2) the focus is inherited by both subformulas  $A$  and  $B$ .

With this process, we may finally come upon an atom  $Q$ , in which case it has to unify with the succedent  $P$ .

$$\frac{Q = P}{\Gamma, [Q] \xrightarrow{f} P} \text{id}$$

Here, we imagine that  $Q = P$  is a unification which instantiates the free variables like  $X$  we have introduced into the proof. Note that the substitution for such variables has to be *global* throughout the partial proof tree. If we wanted to be more formal about this (although I don’t believe it really helps understanding), we would “thread through” a substitution or a set of constraints through all the rules.

The following observation is critical: *if  $Q$  and  $P$  do not unify, then no rule applies here.*

$$\begin{array}{c} \text{no rule if } Q \neq P \\ \Gamma, [Q] \xrightarrow{f} P \end{array}$$

Along the way we have introduced a second judgment form,  $\Gamma \xrightarrow{f} [A]$ , where the focus on the right-hand side. For the example so far, we only

need one rule, where we lose focus.

$$\frac{\Gamma \xrightarrow{f} P}{\Gamma \xrightarrow{f} [P]} \text{blur}$$

Here is a summary of the rules so far:

$$\begin{array}{c} \frac{A \in \Gamma \quad \Gamma, [A] \xrightarrow{f} P}{\Gamma \xrightarrow{f} P} \text{focus}_L \\ \\ \frac{\Gamma, [A(X)] \xrightarrow{f} P}{\Gamma, [\forall x. A(x)] \xrightarrow{f} P} \forall L \qquad \frac{\Gamma, [A] \xrightarrow{f} P \quad \Gamma \xrightarrow{f} [B]}{\Gamma, [B \supset A] \xrightarrow{f} P} \supset L \\ \\ \frac{Q = P}{\Gamma, [Q] \xrightarrow{f} P} \text{id} \qquad \text{no rule if } Q \neq P \\ \qquad \qquad \qquad \frac{}{\Gamma, [Q] \xrightarrow{f} P} \\ \\ \frac{\Gamma \xrightarrow{f} P}{\Gamma \xrightarrow{f} [P]} \text{blur} \end{array}$$

## 4 1+1=2

Now we apply the rules to check the result incrementing 1 in binary form.  
Recall the encoding of the rules

$$\begin{aligned} \Gamma_{\text{inc}} &= \text{inc}(e, b1(e)), \\ &\quad \forall m. \text{inc}(b0(m), b1(m)), \\ &\quad \forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(b1(m), b0(n)) \end{aligned}$$

and the goal sequent, as yet without proof

$$\vdots \\ \Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(b1(e), b0(b1(e)))$$

First we try  $\text{inc}(e, b1(e))$ :

$$\vdots \\ \frac{\Gamma_{\text{inc}}, [\text{inc}(e, b1(e))] \xrightarrow{f} \text{inc}(b1(e), b0(b1(e)))}{\Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(b1(e), b0(b1(e)))} \text{focus}_L$$

We fail immediately, because we are in the situation of the form  $\Gamma, [Q] \xrightarrow{f} P$  where  $Q \neq P$ .

Next we try the second proposition in  $\Gamma_{\text{inc}}$ :

$$\frac{\vdots}{\begin{array}{c} \Gamma_{\text{inc}}, [\forall m. \text{inc}(b0(m), b1(m))] \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \\ \Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \end{array}} \text{focus}_L$$

At this point we use a variable  $M$  for  $m$ , postpone a choice until some unification constraints help us determine a good instantiation of the quantifier.

$$\frac{\vdots}{\begin{array}{c} \Gamma_{\text{inc}}, [\text{inc}(b0(M), b1(M))] \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \\ \Gamma_{\text{inc}}, [\forall m. \text{inc}(b0(m), b1(m))] \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \\ \Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \end{array}} \forall L$$

Again we fail, this time because  $b0(M)$  does not unify with  $b1(e)$ . Third time's a charm:

$$\frac{\vdots}{\begin{array}{c} \Gamma_{\text{inc}}, [\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(b1(m), b0(n))] \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \\ \Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \end{array}} \text{focus}_L$$

Combining two consecutive  $\forall L$  rules, we arrive at

$$\frac{\vdots}{\begin{array}{c} \Gamma_{\text{inc}}, [\text{inc}(M, N) \supset \text{inc}(b1(M), b0(N))] \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \\ \Gamma_{\text{inc}}, [\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(b1(m), b0(n))] \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \\ \Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(b1(e), b0(b1(e))) \end{array}} \forall L \times 2$$

Now we are forced into the  $\supset L$  rule. The incomplete proof state is now

$$\frac{\begin{array}{c} \vdots \\ \Gamma_{\text{inc}}, [\text{inc}(\text{b1}(M), \text{b0}(N))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e}))) \quad \Gamma_{\text{inc}} \xrightarrow{f} [\text{inc}(M, N)] \\ \hline \Gamma_{\text{inc}}, [\text{inc}(M, N) \supset \text{inc}(\text{b1}(M), \text{b0}(N))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e}))) \end{array}}{\Gamma_{\text{inc}}, [\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(\text{b1}(m), \text{b0}(n))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))} \supset L$$

$$\frac{\Gamma_{\text{inc}}, [\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(\text{b1}(m), \text{b0}(n))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))}{\Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))} \text{focus } L$$

At this point, the first branch succeeds with unification, with  $M = \text{e}$  and  $N = \text{b1}(\text{e})$ .

$$\frac{\begin{array}{c} \text{inc}(\text{b1}(M), \text{b0}(N)) = \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e}))) \quad \vdots \\ \hline \Gamma_{\text{inc}}, \text{inc}(\text{b1}(M), \text{b0}(N)) \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e}))) \quad \Gamma_{\text{inc}} \xrightarrow{f} [\text{inc}(M, N)] \end{array}}{\Gamma_{\text{inc}}, [\text{inc}(M, N) \supset \text{inc}(\text{b1}(M), \text{b0}(N))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))} \supset L$$

$$\frac{\Gamma_{\text{inc}}, [\text{inc}(M, N) \supset \text{inc}(\text{b1}(M), \text{b0}(N))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))}{\Gamma_{\text{inc}}, [\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(\text{b1}(m), \text{b0}(n))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))} \forall L \times 2$$

$$\frac{\Gamma_{\text{inc}}, [\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(\text{b1}(m), \text{b0}(n))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))}{\Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))} \text{focus } L$$

This substitution has to be applied globally to the partial proof, which yields

$$\frac{\begin{array}{c} \vdots \\ \hline \Gamma_{\text{inc}}, \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e}))) \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e}))) \quad \Gamma_{\text{inc}} \xrightarrow{f} [\text{inc}(\text{e}, \text{b1}(\text{e}))] \\ \hline \Gamma_{\text{inc}}, [\text{inc}(\text{e}, \text{b1}(\text{e})) \supset \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e}))) \end{array}}{\Gamma_{\text{inc}}, [\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(\text{b1}(m), \text{b0}(n))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))} \supset L$$

$$\frac{\Gamma_{\text{inc}}, [\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(\text{b1}(m), \text{b0}(n))] \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))}{\Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(\text{b1}(\text{e}), \text{b0}(\text{b1}(\text{e})))} \text{focus } L$$

Singling out the remaining subgoal, we can solve this now by focusing on the first of the propositions in  $\Gamma_{\text{inc}}$ . Trying to focus on any other one of the

antecedents will fail.

$$\frac{\frac{\frac{\Gamma_{\text{inc}}, [\text{inc}(e, b1(e))] \xrightarrow{f} \text{inc}(e, b1(e))}{\Gamma_{\text{inc}} \xrightarrow{f} \text{inc}(e, b1(e))} \text{id}}{\Gamma_{\text{inc}} \xrightarrow{f} [\text{inc}(e, b1(e))]} \text{focus}_L}{\Gamma_{\text{inc}} \xrightarrow{f} [\text{inc}(e, b1(e))]} \text{blur}_R$$

Notice that in the example we could have added many many other propositions in the antecedent and it would not have changed the outcome as long as the eventual proposition at the end of the chain of quantifiers and implications is not of the form  $\text{inc}(\_, \_)$ . In an implementation of a logic programming language such as Prolog we compile the programs (that is, the antecedents) so that we have direct access to those defining the particular predicate in the query (that is, the succedent).

## 5 Horn Clauses

For which fragment of the logic does this proof search strategy work? Answering this question will give us (a slight generalization) of *Horn clauses*, and those are the only ones allowed in backward chaining logic programming languages like Prolog.

A key seems to be that the antecedents all have noninvertible left rules that can be chained together. Similarly, we would like the succedent to have noninvertible right rules, again so we can chain them (not visible in this example). Remembering polarities: those propositions with invertible right rules are *negative* and have noninvertible left rules, while those with invertible left rules have noninvertible right rules are *positive*. We write  $D^-$  for the negative propositions in the antecedent<sup>1</sup>, and  $G^+$  for the positive proposition in the succedent. We concentrate on the core propositions in red and leave out some additional propositions, which the theory would predict are compatible with backward chaining.

Program formulas	$D^- ::= P^- \mid G^+ \supset D^- \mid \forall x. D^-(x) \mid D_1^- \wedge D_2^- \mid \top$
Programs	$\Gamma^- ::= \cdot \mid \Gamma^-, D^-$
Goal formulas	$G^+ ::= \downarrow P^- \mid G_1^+ \wedge G_2^+ \mid \top \mid \exists x. G^+(x) \mid G_1^+ \vee G_2^+ \mid \perp$

---

<sup>1</sup> $D$  stands for “definitive clauses” from the early days of Prolog.

There may be a couple of surprises here. One is that conjunction is both positive and negative. This is because there two forms of left rules for conjunction, one that decomposes  $A \wedge B$  to  $A, B$  (which is invertible, and therefore positive) and one that extracts either  $A$  or  $B$  (which is noninvertible and therefore negative). We have three judgment forms

$$\begin{array}{ll} \Gamma^- \xrightarrow{f} P^- & \text{stable sequent} \\ \Gamma^-, [D^-] \xrightarrow{f} P^- & \text{left focus} \\ \Gamma^- \xrightarrow{f} [G^+] & \text{right focus} \end{array}$$

It is also important to consider that we *left out*, which is (syntactically) very little: program formulas  $D^-$  cannot be of the form  $\uparrow G^+$ , and goal formulas  $G^+$  can neither be positive atoms  $P^+$  nor of the form  $\downarrow D^-$  (except  $\downarrow P^-$ ).

$$\begin{array}{ll} \text{Program formulas} & D^- ::= \dots \mid \uparrow G^- \\ \text{Goal formulas} & G^+ ::= \dots \mid P^+ \mid \downarrow D^- \end{array}$$

By forcing goals to be almost entirely positive and programs to be entirely negative, no inversion will ever be applied. The fact that we omitted positive atoms means chaining is always backwards (see next lecture). Essentially, backward chaining logic programming arises from a pure backward chaining interpretation of intuitionistic logic.

Another interesting aspect is that Horn clauses are so restricted that classical and intuitionistic logic coincide on them. That is, a goal  $P^-$  is provable from a program  $\Gamma^-$  in classical logic if and only if the sequent is provable in intuitionistic logic. So at least in logic programming, there shouldn't be any arguments between intuitionists and classical logicians.

We now restate the rules for core Horn clauses, using our new notation. Also, for existential goals, we introduce the  $\exists R$  rule. For both  $\exists R^*$  and  $\forall L^*$  we need a globally fresh variable  $X$ ; the superscript \* is supposed to be a reminder of this condition.

$$\begin{array}{c}
\frac{D^- \in \Gamma^- \quad \Gamma, [D^-] \xrightarrow{f} P^-}{\Gamma^- \xrightarrow{f} P^-} \text{ focus}_L \\
\\
\frac{\Gamma^-, [D^-(X)] \xrightarrow{f} P^-}{\Gamma^-, [\forall x. D^-(x)] \xrightarrow{f} P^-} \forall L^* \quad \frac{\Gamma^-, [D^-] \xrightarrow{f} P^- \quad \Gamma^- \xrightarrow{f} [G^+]}{\Gamma^-, [G^+ \supset D^-] \xrightarrow{f} P^-} \supset L \\
\\
\frac{Q^- = P^-}{\Gamma^-, [Q^-] \xrightarrow{f} P^-} \text{ id} \quad \text{no rule if } Q^- \neq P^- \\
\\
\frac{\Gamma^- \xrightarrow{f} [G_1^+] \quad \Gamma^- \xrightarrow{f} [G_2^+]}{\Gamma^- \xrightarrow{f} [G_1^+ \wedge G_2^+]} \wedge R \quad \frac{}{\Gamma^- \xrightarrow{f} [\top]} \top R \\
\\
\frac{\Gamma^- \xrightarrow{f} [G(X)]}{\Gamma^- \xrightarrow{f} [\exists x. G^+(x)]} \exists R^* \quad \frac{\Gamma^- \xrightarrow{f} P^-}{\Gamma^- \xrightarrow{f} [\downarrow P^-]} \text{ blur}
\end{array}$$

## 6 Writing a Meta-Interpreter

Now that we have rewritten the foundation of logic programming in propositional form, can we take advantage of this for an implementation? An ideal language for such an implementation would already support unification and backtracking, since both of these are at the heart of the computation-as-proof-search paradigm. We don't have to look far: let's use Prolog! This means we are writing what is called a *meta-circular interpreter*: we provide the semantics of a language in itself. There can be some arguments whether this counts as "definitional" since, for example, the meta-interpreter will not explain the details of unification because they are just inherited from the metalanguage. Nevertheless, we can modify the semantics of chaining by modifying the meta-interpreter, and we will play through an example of this.

Now consider a program  $\Gamma^-$  and a query  $P^-$ , which should proceed by searching for a proof of  $\Gamma^- \xrightarrow{f} P^-$ . In the remainder of this section we will just omit the polarity annotations, remembering the program clauses  $D$  and atoms  $P, Q$  are negative goals  $G$  are positive.

We begin with program representation, that is, the representation of  $\Gamma$ . The idea is that every element  $D$  of  $\Gamma$  is represented in Prolog as a fact `prog(D)`. For example,

```
prog(inc(e, b1(e))).
```

This clause contains no variables. For the next one,  $\forall m. \text{inc}(b0(m), b1(m))$ , we have to determine how to represent the quantifier. Unfortunately, this is difficult in Prolog because it leaves quantifiers implicit and considers free variables in the program to be universally quantified. We exploit this, by instantiating  $m$  with a new free variable  $M$  and stating

```
prog(inc(b0(M), b1(M))).
```

For Horn clauses as we have defined them, it is always possible to move the universal quantifier in program formulas  $D$  to the outside.

Finally, the last proposition in  $\Gamma_{\text{inc}}$

$$\forall m. \forall n. \text{inc}(m, n) \supset \text{inc}(b1(m), b0(n))$$

requires an implication. We write this in prefix form,  $G \supset D$  as `imp(G, D)`, so we don't get confused between the language we are interpreting `imp(G, D)` and the language in which we implement it in, whose implication would be written as `D :- G`.

```
prog(imp(inc(M, N), inc(b1(M), b0(N)))).
```

Again, the quantified variables become free variables.

A priori, we could think the proof search would be through a predicate `solve(Gamma, G)`. But the program  $\Gamma$  never changes during search in the Horn fragment, so it is sufficient to consider just `solve(G)`. The first two clauses should be clear: they implement  $\wedge R$  and  $\top R$ :

```
solve(true).
solve(and(G1, G2)) :- solve(G1), solve(G2).
```

The third possibility is an atom. We did not complicate the syntax with an explicit atom constructor (perhaps we should have ...), so we have a predicate `atm/1` that recognizes atoms. In our example:

```
atm(inc(_, _)).
```

After we have recognized that we are trying to prove an atom, we have to nondeterministically select a program clause  $D$  and then see if  $\Gamma, [D] \xrightarrow{f} P$ . Selecting  $D$  takes place by calling `prog(D)`, and the  $\Gamma, [D] \xrightarrow{f} P$  will be implemented by `focus(D, P)`.  $\Gamma$  can remain implicit, because it is represented as facts in Prolog as explained before.

```
solve(P) :- atm(P), prog(D), focus(D, P).
```

The `focus/2` predicate now distinguishes the cases for its first argument. Since quantifiers are implicit, at this point this only pertains to implications and atoms. For atoms, we unify them and make sure no clause applies if they don't unify.

```
focus(Q,P) :- atm(Q), Q = P.
```

For implications, we continue to focus on  $D$  first, and if that eventually succeeds, we solve the subgoal.

```
focus(imp(G,D),P) :- focus(D,P), solve(G).
```

At this point, we already have a complete meta-interpreter for the Horn clause fragment with prefixed quantifiers. Here is the summary.

```
solve(true).
solve(and(G1,G2)) :- solve(G1), solve(G2).
solve(P) :- atm(P), prog(D), focus(D, P).
focus(Q,P) :- atm(Q), Q = P.
focus(imp(G,D),P) :- focus(D,P), solve(G).
```

Our example of binary addition:

```
atm(inc(_,_)).
prog(inc(e,b1(e))).
prog(inc(b0(M),b1(M))).
prog(imp(inc(M,N),inc(b1(M),b0(N)))).
```

We can now exercise the meta-interpreter, again using free variables in the query to model existentially quantified variables in the query.

```
| ?- solve(inc(b1(e),N)).
N = b0(b1(e)) ? ;
```

```
(1 ms) no
| ?- solve(inc(M,b1(e))).

M = e ? ;
M = b0(e) ? ;
no
```

With the second example we also see that Prolog backtracking implements backtracking in our small Horn clause language.

## 7 Repairing Unsound Unification

One unfortunate consequence of using Prolog as our implementation language is that we inherit its unsound unification. This means as a logic-based proof search engine, our meta-interpreter has a severe bug. Fortunately, Prolog also provides us with the means to fix it.

As an example, consider the query which should have no solution: there is no  $m$  such that  $\text{inc}(\text{b0}(m), \text{b1}(\text{b0}(m)))$ ! But instead of failing, it uses the second clause and incorrectly succeeds in unifying  $M = \text{b0}(M)$ , creating a circular term  $M = \text{b0}(\text{b0}(\dots))$

```
?- solve(inc(b0(M), b1(b0(M)))).
cannot display cyclic term for M ? ;
no
```

If we want to make the program sound, we have to scour the meta-interpreter for (a) explicit calls to unification, or (b) implicit unifications which arise from repeated variables in the heads of clauses. We see there are no repeated variables, and the only explicit call to unification is in the case of atoms. So we can just replace this with a call to a library predicate that implements sound unification.

```
solve(true).
solve(and(G1,G2)) :- solve(G1), solve(G2).
solve(P) :- atm(P), prog(D), focus(D, P).
% focus(Q,P) :- atm(Q), Q = P.      % unsound unification
focus(Q,P) :- atm(Q), unify_with_occurs_check(Q,P).
focus(imp(G,D),P) :- focus(D,P), solve(G).
```

Now the previous query fails, as we had hoped.

```
?- solve(inc(b0(M), b1(b0(M)))).
```

no

## 8 Further Variants of the Semantics

We can now add the remaining connectives from our definitions of positive and negative propositions, leaving only the quantifiers implicit. Note that there are two clauses for disjunctive goals, and two clauses for conjunctive programs. We could also have implemented this using a goal `A ; B` in Prolog, but we prefer to keep the meta-language constructs simple.

Just to show what can be done with a metainterpreter, we also add a minimal tracing facility. We do this via a predicate `display/1` which outputs its argument, and `nl/0` which outputs a newline. Note that `fail` is necessary so that Prolog backtracks after it has printed the goal and uses the other clauses for `solve/1` to actually solve the goal. Other variations on this can be easily imagined.

```
% solve(G) succeeds if Gamma --> G
% focus(D,Q) succeeds if Gamma, [D] --> Q
solve(G) :- display(G), nl, fail.

solve(true).
solve(and(G1,G2)) :- solve(G1), solve(G2).
solve(or(G1,G2)) :- solve(G1).
solve(or(G1,G2)) :- solve(G2).
% no clause for solve(false)
solve(P) :- atm(P), prog(D), focus(D,P).

focus(imp(G,D),P) :- focus(D,P), solve(G).
focus(and(D1,D2),P) :- focus(D1, P).
focus(and(D1,D2),P) :- focus(D2, P).
focus(Q,P) :- atm(Q), unify_with_occurs_check(Q,P).
```

For example, the following query prints the three atomic goals it encounters in sequence (due to the carry bit), in each case with fresh internally named existential variables.

```
?- solve(inc(b1(b1(e)),N)).  
inc(b1(b1(e)),_283)  
inc(b1(e),_302)  
inc(e,_315)  
  
N = b0(b0(b1(e))) ? ;  
  
no
```

As a final possibility, consider how you might instrument the interpreter to not just present an answer substitution, but a proof term. The `solve/1` predicate would then be generalized to `solve/2` and we would ask, for example

```
solve(M, inc(e, N))
```

for some goal which would not only show  $N = b1(e)$  but also a proof  $M : inc(e, b1(e))$  that was found by the interpreter.

# Lecture Notes on Datalog

15-317: Constructive Logic  
Frank Pfenning

Lecture 18  
November 2, 2017

## 1 Introduction

In the previous lecture we have seen backward chaining from a logical perspective, and how this can be seen as a foundation for backward-chaining logic programming languages like Prolog.

In this lecture we take a small step sideways: instead of considering all atoms to be *negative* we consider all atoms *positive*. This has a rather drastic impact on the operational behavior of proof search, leading to *forward-chaining logic programming*. This is also called *bottom-up logic programming*, although the direction is strangely reversed from the way we consider the proof construction process.

## 2 Reading Inference Rules from Premises to Conclusion

Mostly over the last serious of lectures, we read inference rules by looking at the conclusion first and then the premises. This was so, because that is the direction of proof construction. In fact, the sequent calculus was specifically engineered by Gentzen to have this property!

Now we will read inference rules starting with the premises. For example, assume we would like to calculate the path relation in an undirected graph, where we say there is a path from vertex  $x$  to  $y$  if there is a sequence of vertices  $x = x_0, x_1, \dots, x_n = y$  such that all  $x_i$  and  $x_{i+1}$  are connected by an edge. For simplicity, let us say that  $n \geq 1$ .

We represent the vertices of a graph by constants, and the edge relation with a predicate  $\text{edge}(x, y)$  if there is an edge from  $x$  to  $y$ . Here is a specification of the path relation:

$$\frac{\text{edge}(x, y)}{\text{edge}(y, x)} \text{ sym} \quad \frac{\text{edge}(x, y)}{\text{path}(x, y)} \text{ ep} \quad \frac{\text{path}(x, y) \quad \text{path}(y, z)}{\text{path}(x, z)} \text{ trans}$$

The first rule (sym) expresses we are working over an undirected graph. The second (ep) expresses that an edge represents a valid path, and the third that the path relation is transitive.

Read from the conclusion to the premises, backward logic programming search over this specification is useless. Even just the rule sym rule will lead to an infinite loop, and the trans rule has an unknown  $y$  in the premise even if  $x$  and  $z$  are known in the conclusion.

Read from the premises to the conclusion, however, this is a decent program *if we avoid re-deriving facts we already know*. After while, this program must terminate because there are at most  $O(n^2)$  facts of the form  $\text{edge}(x, y)$  and  $\text{path}(x, y)$  that could be derived. When inference reaches the point where any additional inference only infers facts we already know, we say the program has *reached saturation* and it halts. At this point we can answer any specific query simply by looking it up in the collection of derived facts, usually called the *database*.

### 3 Saturation

As another example, we first consider the usual bottom-up specifications of  $\text{even}(n)$  and  $\text{odd}(n)$  for unary numbers.

$$\frac{}{\text{even}(z)} \text{ ev}_z \quad \frac{\text{odd}(N)}{\text{even}(s(N))} \text{ ev}_s \quad \frac{\text{even}(N)}{\text{odd}(s(N))} \text{ od}_s$$

Reading these from the premise to the conclusion does not work: these rules would create an unbounded database with facts

$$\text{even}(z), \text{odd}(s(z)), \text{even}(s(s(z))), \dots$$

But if we view these as *introduction rules* we can derive *elimination rules* that work in the other direction, using what we already know!

$$\frac{\text{even}(s(N))}{\text{odd}(N)} \text{ ev}'_s \quad \frac{\text{odd}(s(N))}{\text{even}(N)} \text{ od}'_s \quad \frac{\text{odd}(z)}{C} \text{ od}'_z$$

Note that there is no rule for  $\text{even}(z)$  because we cannot extract any information from that: the rule  $\text{ev}_z$  has no premises. In the last rule we have derived a contradiction, which is manifest in being able to conclude any proposition  $C$ . If we want to use this as a program, we have to use a specific proposition. Unfortunately,  $\uparrow\perp$  is not part of the chaining fragment (for good reason), so we use a new atom  $\text{no}$ , in the best tradition of the Prolog top level.

If we want to know if, say, the fact  $\text{even}(\text{s}(\text{s}(\text{s}(z))))$  is *consistent with* the definition of the predicate, we assert it in the database of facts and saturate the database using forward inference. Because of the simple nature of these rules, our hand is forced at each point of inference, and we obtain the following saturated database:

$\text{even}(\text{s}(\text{s}(\text{s}(z)))), \text{odd}(\text{s}(\text{s}(z))), \text{even}(\text{s}(z)), \text{odd}(z), \text{no}$

This tells us that asserting that 3 is even is inconsistent with the definition of evenness. On the other hand, if we assert  $\text{even}(\text{s}(\text{s}(z)))$ , we learn:

$\text{even}(\text{s}(\text{s}(z))), \text{odd}(\text{s}(z)), \text{even}(z)$

This database is now saturated and there is no contradiction, so the assertion that 2 is even is consistent with the definition of evenness.

## 4 Forward Chaining

Our translation from rules to propositions leads us to the following propositions representing the downward-reading rules for even and odd numbers:

$$\begin{aligned}\Gamma_{eo} = & \forall n. \text{even}(\text{s}(n)) \supset \text{odd}(n), \\ & \forall n. \text{odd}(\text{s}(n)) \supset \text{even}(n), \\ & \text{odd}(z) \supset \text{no}\end{aligned}$$

We then ask, for example,

$$\Gamma_{eo}, \text{even}(\text{s}(\text{s}(\text{s}(z)))) \longrightarrow \text{no}$$

to find out if the fact that 3 is even is consistent with the knowledge in  $\Gamma_{eo}$ .

This search is now *the exact opposite of goal directed*, let's call it *database directed*. We ignore the succedent ( $\text{no}$ ) and saturate the database, which are the atoms in the antecedents. Only once we have saturated the database, do we even look at the succedent and see if it is a fact in the database (or,

more generally, can be proven from the database directly without further forward chaining).

For this intuition to work, we have to start by instantiating  $n$  in the first proposition with  $s(s(z))$  and then use the implication left rule to conclude  $\text{odd}(s(s(z)))$ . It is this process we call *forward chaining*. To formalize this, we first recall the language and rules for *backward chaining*.

**Backward chaining fragment:** all atoms are negative, and the only polarity shift has the form  $\downarrow P^-$ . So far, we have only shown the rules for the connectives in red.

$$\begin{array}{ll} \text{Program formulas} & D^- ::= P^- \mid G^+ \supset D^- \mid \forall x. D^-(x) \mid D_1^- \wedge D_2^- \mid \top \\ \text{Programs} & \Gamma^- ::= \cdot \mid \Gamma^-, D^- \\ \text{Goal formulas} & G^+ ::= \downarrow P^- \mid G_1^+ \wedge G_2^+ \mid \top \mid \exists x. G^+(x) \mid G_1^+ \vee G_2^+ \mid \perp \end{array}$$

**Forward chaining fragment:** all atoms are positive, and the only polarity shift has the form  $\uparrow P^+$ .

$$\begin{array}{ll} \text{Program formulas} & D^- ::= \uparrow P^+ \mid G^+ \supset D^- \mid \forall x. D^-(x) \mid D_1^- \wedge D_2^- \mid \top \\ \text{Database} & \Gamma ::= \cdot \mid \Gamma, D^- \mid \Gamma, P^+ \\ \text{Goal formulas} & G^+ ::= P^+ \mid G_1^+ \wedge G_2^+ \mid \top \mid \exists x. G^+(x) \mid G_1^+ \vee G_2^+ \mid \perp \end{array}$$

The antecedents now mix the program formulas  $D^-$  (sometimes called the IDB) and the database facts  $P^+$  (sometimes called the EDB), while the succedent is always positive since negative atoms are not part of this fragment. We have the following three judgments

Backward Chaining		Forward Chaining
$\Gamma^- \xrightarrow{f} P^-$	stable sequent	$\Gamma \xrightarrow{f} C^+$
$\Gamma^-, [D^-] \xrightarrow{f} P^-$	left focus	$\Gamma, [D^-] \xrightarrow{f} C^+$
$\Gamma^- \xrightarrow{f} [G^+]$	right focus	$\Gamma \xrightarrow{f} [G^+]$

The rules for the connectives remain the same, with the exception of the order of premises in the  $\supset L$  rule.<sup>1</sup> We also remove the rules concerned with negative atoms and add those for positive ones.

<sup>1</sup>It is possible that in a combined forward/backward chaining language, there should be two forms of implication, presenting the premises in different order:  $A \supset B$  proves  $A$  before assuming  $B$ , and  $B \subset A$  assumes  $B$  before proving  $A$ . This first one would be employed in forward chaining, the second in backward chaining. Of course, logically the two are the same, but not operationally when viewed from the computation-as-proof-search perspective.

$$\begin{array}{c}
\frac{D^- \in \Gamma \quad \Gamma, [D^-] \xrightarrow{f} C^+}{\Gamma \xrightarrow{f} C^+} \text{ focus}_L \quad \frac{\Gamma, P^+ \xrightarrow{f} C^+}{\Gamma, [\uparrow P^+] \xrightarrow{f} C^+} \uparrow_L \\
\\
\frac{\Gamma, [D^-(X)] \xrightarrow{f} C^+}{\Gamma, [\forall x. D^-(x)] \xrightarrow{f} C^+} \forall L^* \quad \frac{\Gamma \xrightarrow{f} [G^+] \quad \Gamma, [D^-] \xrightarrow{f} C^+}{\Gamma, [G^+ \supset D^-] \xrightarrow{f} C^+} \supset L \\
\\
\text{no longer applicable} \left[ \begin{array}{ll} \frac{Q^- = P^-}{\Gamma, [Q^-] \xrightarrow{f} P^-} \text{id}^- & \text{no rule if } Q^- \neq P^- \\ & \Gamma, [Q^-] \xrightarrow{f} P^- \end{array} \right] \\
\\
\frac{Q^+ \in \Gamma \quad Q^+ = P^+}{\Gamma \xrightarrow{f} [P^+]} \text{id}^+ \quad \text{no rule if } Q^+ \neq P^+ \text{ for all } Q^+ \in \Gamma \quad \frac{}{\Gamma \xrightarrow{f} [\top]} \top R \\
\\
\frac{\Gamma \xrightarrow{f} [G_1^+] \quad \Gamma \xrightarrow{f} [G_2^+]}{\Gamma \xrightarrow{f} [G_1^+ \wedge G_2^+]} \wedge R \quad \frac{}{\Gamma \xrightarrow{f} [\top]} \top R \\
\\
\frac{\Gamma \xrightarrow{f} [G(X)]}{\Gamma \xrightarrow{f} [\exists x. G^+(x)]} \exists R^* \quad \text{no longer applicable} \left[ \begin{array}{l} \frac{\Gamma^- \xrightarrow{f} P^-}{\Gamma^- \xrightarrow{f} [\downarrow P^-]} \downarrow R \\ \frac{\Gamma^- \xrightarrow{f} [\downarrow P^-]}{} \end{array} \right] \\
\\
\frac{\Gamma, [D_1^-] \xrightarrow{f} C^+}{\Gamma, [D_1^- \wedge D_2^-] \xrightarrow{f} C^+} \wedge L_1 \quad \frac{\Gamma, [D_2^-] \xrightarrow{f} C^+}{\Gamma, [D_1^- \wedge D_2^-] \xrightarrow{f} C^+} \wedge L_2 \quad \text{no rule for } \top L \\
\\
\frac{\Gamma \xrightarrow{f} [G_1^+]}{\Gamma \xrightarrow{f} [G_1^+ \vee G_2^+]} \vee R_1 \quad \frac{\Gamma \xrightarrow{f} [G_2^+]}{\Gamma \xrightarrow{f} [G_1^+ \vee G_2^+]} \vee R_2 \quad \text{no rule for } \perp R
\end{array}$$

Figure 1: Forward chaining fragment of Horn logic

Let's observe these rules in action on our program and goal, where all atoms are positive. We have added shifts on the right-hand side of implications during polarization of the propositions.

$$\begin{aligned}\Gamma_{eo} = \forall n. \text{even}(s(n)) &\supset \uparrow \text{odd}(n), \\ \forall n. \text{odd}(s(n)) &\supset \uparrow \text{even}(n), \\ \text{odd}(z) &\supset \uparrow \text{no}\end{aligned}$$

$$\Gamma_{eo}, \text{even}(s(s(s(z)))) \xrightarrow{f} \text{no}$$

Say we focus on the first proposition in  $\Gamma_{eo}$ .

$$\frac{\vdots \quad \vdots}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))) \xrightarrow{f} [\text{even}(s(N))] \quad \Gamma_{eo}, \text{even}(s(s(s(z)))), [\uparrow \text{odd}(N)] \xrightarrow{f} \text{no}}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))), [\text{even}(s(N)) \supset \uparrow \text{odd}(n)] \xrightarrow{f} \text{no}}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))), [\forall n. \text{even}(s(n)) \supset \uparrow \text{odd}(n)] \xrightarrow{f} \text{no}}{\supset L}} \forall L}}$$

Now we match  $\text{even}(s(s(s(z))))$  against  $\text{even}(s(N))$  which succeeds with substitution  $N = s(s(z))$ , which is applied globally to the partial proof which then looks as follows:

$$\frac{\vdots \quad \vdots}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))) \xrightarrow{f} [\text{even}(s(s(s(z))))] \quad \Gamma_{eo}, \text{even}(s(s(s(z)))), [\uparrow \text{odd}(s(s(z)))] \xrightarrow{f} \text{no}}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))), [\text{even}(s(s(s(z))) \supset \uparrow \text{odd}(n))] \xrightarrow{f} \text{no}}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))), [\forall n. \text{even}(s(n)) \supset \uparrow \text{odd}(n)] \xrightarrow{f} \text{no}}{\supset L}} \forall L}} \supset L}$$

In the right branch of the proof we now lose focus, and we have reached a stable sequent, adding another fact to the database.

$$\frac{\vdots \quad \vdots}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))) \xrightarrow{f} [\text{even}(s(s(s(z))))] \quad \frac{\Gamma_{eo}, \text{even}(s(s(s(z)))), \text{odd}(s(s(z))) \xrightarrow{f} \text{no}}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))), [\uparrow \text{odd}(s(s(z)))] \xrightarrow{f} \text{no}}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))), [\text{even}(s(s(s(z))) \supset \uparrow \text{odd}(n))] \xrightarrow{f} \text{no}}{\frac{\Gamma_{eo}, \text{even}(s(s(s(z)))), [\forall n. \text{even}(s(n)) \supset \uparrow \text{odd}(n)] \xrightarrow{f} \text{no}}{\supset L}} \forall L}} \uparrow L}} \supset L}$$

This will continue until we add  $\text{odd}(z)$  and then  $\text{no}$  to the database. At this point the goal looks like

$$\begin{array}{c} \vdots \\ \Gamma_{\text{eo}}, \dots, \text{no} \xrightarrow{f} \text{no} \end{array}$$

Recall that all atoms are positive, so we can now focus on the succedent and complete the proof.

$$\frac{\Gamma_{\text{eo}}, \dots, \text{no} \xrightarrow{f} [\text{no}]}{\Gamma_{\text{eo}}, \dots, \text{no} \xrightarrow{f} \text{no}} \text{id}^+ \text{ focus}_R$$

We can also see what happens if we focus on the “wrong” antecedent, that is, one that we cannot use with forward chaining.

$$\frac{\text{fails} \quad \vdots}{\Gamma_{\text{eo}}, \text{even}(\text{s}(\text{s}(\text{s}(z)))) \xrightarrow{f} [\text{odd}(\text{s}(N))] \quad \Gamma_{\text{eo}}, \text{even}(\text{s}(\text{s}(\text{s}(z))))), [\uparrow \text{even}(N)] \xrightarrow{f} \text{no}} \supset L$$

$$\frac{\Gamma_{\text{eo}}, \text{even}(\text{s}(\text{s}(\text{s}(z)))), [\text{odd}(\text{s}(N)) \supset \uparrow \text{even}(N)] \xrightarrow{f} \text{no}}{\Gamma_{\text{eo}}, \text{even}(\text{s}(\text{s}(\text{s}(z)))), [\forall n. \text{odd}(\text{s}(n)) \supset \uparrow \text{even}(n)] \xrightarrow{f} \text{no}} \forall L$$

Experience shows that, generally, the reduction in the search space with forward chaining is not quite as drastic as with backward chaining. This is because there is only one goal (the succedent) but many clauses in the database (the antecedents). Nevertheless, there are many algorithms more easily described with forward chaining than with backward chaining. We will show one in Section 6 on unification.

## 5 Comparing Backward and Forward Chaining

We return to a simple example from last lecture

$$a, a \supset b, b \supset c \longrightarrow c$$

If we polarize all atoms negatively, we obtain

$$a^-, \downarrow a^- \supset b^-, \downarrow b^- \supset c^- \xrightarrow{f} c^-$$

where the shift operators binds tightest.

Without backward chaining there are different proofs. In particular, we could apply  $\supset L$  to  $a \supset b$ , or to  $b \supset c$ . With backward chaining, there is only one possible proof: at each choice point, when we focus, only on possibility will succeed and the others will fail immediately. You should convince yourself that this is the case. We define  $\Gamma_0 = (a^-, \downarrow a^- \supset b^-, \downarrow b^- \supset c^-)$

$$\begin{array}{c}
 \frac{}{\Gamma_0, [a^-] \xrightarrow{f} a^-} \text{id}^- \\
 \frac{}{\Gamma_0, [b^-] \xrightarrow{f} b^-} \text{id}^- \quad \frac{\Gamma_0 \xrightarrow{f} a^-}{\Gamma_0 \xrightarrow{f} [\downarrow a^-]} \downarrow R \\
 \frac{\Gamma_0, [b^-] \xrightarrow{f} b^- \quad \Gamma_0 \xrightarrow{f} [\downarrow a^-]}{\Gamma_0, [\downarrow a^- \supset b^-] \xrightarrow{f} b^-} \supset L \\
 \frac{}{\Gamma_0, [c^-] \xrightarrow{f} c^-} \text{id}^- \quad \frac{\Gamma_0 \xrightarrow{f} b^-}{\Gamma_0 \xrightarrow{f} [\downarrow b^-]} \downarrow R \\
 \frac{\Gamma_0, [c^-] \xrightarrow{f} c^- \quad \Gamma_0 \xrightarrow{f} [\downarrow b^-]}{\Gamma_0, [\downarrow b^- \supset c^-] \xrightarrow{f} c^-} \supset L \\
 \frac{}{\Gamma_0 \xrightarrow{f} c^-} \text{id}^-
 \end{array}$$

Going back to our sequent

$$a, a \supset b, b \supset c \longrightarrow c$$

if we polarize all the atoms positively, in preparation for forward chaining, we get

$$a^+, a^+ \supset \uparrow b^+, b^+ \supset \uparrow c^+ \xrightarrow{f} c^+$$

Now we must focus on  $a^+ \supset \uparrow b^+$  first, then  $b^+ \supset \uparrow c^+$ , then  $c^+$ . All other attempts at focusing will either fail, or lead conclude a fact that is already

in the database. We abbreviate  $\Gamma_1 = (a^+, a^+ \supset b^+, b^+ \supset c^+)$

$$\begin{array}{c}
 \frac{}{\Gamma_1, b^+, c^+ \xrightarrow{f} [c^+]} \text{id}^+ \quad \frac{}{\Gamma_1, b^+, c^+ \xrightarrow{f} c^+} \text{focus}_R \\
 \frac{}{\Gamma_1, b^+ \xrightarrow{f} [b^+]} \text{id}^+ \quad \frac{\Gamma_1, b^+, c^+ \xrightarrow{f} c^+}{\Gamma_1, b^+, [\uparrow c^+] \xrightarrow{f} c^+} \uparrow L \\
 \frac{\text{id}^+ \quad \Gamma_1, b^+, [\uparrow c^+] \xrightarrow{f} c^+}{\Gamma_1, b^+, [b^+ \supset \uparrow c^+] \xrightarrow{f} c^+} \supset L \\
 \frac{a^+ \in \Gamma \quad \Gamma_1 \xrightarrow{f} [a^+]}{\Gamma_1 \xrightarrow{f} [a^+ \supset b^+]} \text{id}^+ \quad \frac{\Gamma_1, b^+ \xrightarrow{f} c^+ \quad \Gamma_1, [\uparrow b^+] \xrightarrow{f} c^+}{\Gamma_1, b^+ \supset [\uparrow b^+] \xrightarrow{f} c^+} \uparrow L \\
 \frac{}{\Gamma_1, [\uparrow b^+] \xrightarrow{f} c^+} \supset L \\
 \frac{\Gamma_1, [a^+ \supset b^+] \xrightarrow{f} c^+}{\Gamma_1 \xrightarrow{f} c^+} \text{focus}_L
 \end{array}$$

## 6 Unification

As a major and significant example of forward chaining, which is similar to many realistic applications of Datalog, we use *unification* itself. So far, we have just treated it informally, despite its complexity.

We describe the algorithm by a set of rules concerning a predicate  $t \doteq s$  for (first-order) terms  $t$  and  $s$ . This set of rules can be translated to a collection of propositions  $\Gamma_u$  where all atoms are positive. We assert a new equality, adding it as an antecedent, and then saturate the database. If it produces no, then the new equality is inconsistent with all the information we already had. Otherwise, the new saturated database represents the “solution” and shows consistency.

We begin with two rules that compare the function symbol at the head of the two terms. We write  $\bar{t}$  for a sequence of terms.

$$\frac{f(\bar{t}) \doteq f(\bar{s})}{\bar{t} \doteq \bar{s}} \text{con}_= \quad \frac{f(\bar{t}) \doteq g(\bar{s}) \quad f \neq g}{\text{no}} \text{con}_\neq$$

The first rule expresses that if  $f(\bar{t})$  is equal to  $f(\bar{s})$  then the sequences of arguments must be equal. This means that function symbols are “uninterpreted”: they are used as data constructors, not to stand for arbitrary mathematical functions.

The second rule expresses that if the data constructors are different then the terms are not equal. In other words, if we know they are equal, then this is a contradiction.

These two rules also capture constants, since we think of a constant  $c$  and  $c()$ , with the empty sequence of arguments.

Now we need four rules for comparing sequences of arguments.

$$\begin{array}{c} \frac{(t, \bar{t}) \doteq (s, \bar{s}) \text{ seq}_{=1}}{t \doteq s} \quad \frac{(t, \bar{t}) \doteq (s, \bar{s}) \text{ seq}_{=2}}{\bar{t} \doteq \bar{s}} \\[10pt] \frac{() \doteq (s, \bar{s}) \text{ seq}_{\neq 1}}{\text{no}} \quad \frac{(t, \bar{t}) \doteq () \text{ seq}_{\neq 2}}{\text{no}} \end{array}$$

Note that there is not rule for  $() \doteq ()$ , because such an equality contains no information to extract.

At this point we have enough rules to decide *equality*, but not yet enough to implement unification. Consider

$$f(X, X) \doteq f(c(), d())$$

This problem must fail, since  $X$  cannot be equal to  $c()$  and  $d()$  simultaneously, but the rules so far do not account for this. The simple device of stating symmetry and transitivity of equality will solve this particular problem.

$$\begin{array}{c} \frac{t \doteq s \text{ sym}}{s \doteq t} \quad \frac{t \doteq s \quad s \doteq r \text{ trans}}{t \doteq r} \end{array}$$

Now we deduce:

$$\begin{array}{ll} f(X, X) \doteq f(c(), d()) & \text{given} \\ (X, X) \doteq (c(), d()) & \text{by rule con}_= \\ X \doteq c() & \text{by rule seq}_{=1} \\ (X) \doteq (d()) & \text{by rule seq}_{=2} \\ X \doteq d() & \text{by rule seq}_{=1} \\ c() \doteq X & \text{by rule sym} \\ c() \doteq d() & \text{by rule trans} \\ \text{no} & \text{by rule con}_{\neq} \end{array}$$

One could make these rules more efficient, for example, by restricting some terms in symmetry and transitivity to be variables.

At this point we have arrived at Prolog-style unification. Unfortunately, we know that this is unsound, because an equation such as

$$X \doteq f(X)$$

is not recognized as inconsistent. We can incorporate this by adding some rules for the occurs-check that discover such inconsistencies.

$$\begin{array}{c} X \doteq f(\bar{t}) \\ \hline X \notin f(\bar{t}) \end{array} \quad \begin{array}{c} X \notin X \\ \hline \text{no} \end{array}$$

$$\begin{array}{c} X \notin f(\bar{t}) \quad X \notin (t, \bar{t}) \quad X \notin (t, \bar{t}) \\ \hline X \notin \bar{t} \quad X \notin t \quad X \notin \bar{t} \end{array}$$

$$\begin{array}{c} X \notin Y \quad Y \doteq t \\ \hline X \notin t \end{array}$$

The last rule is necessary to obtain contradictions from problems such as

$$X = f(Y), Y = g(X)$$

With a few optimizations, these rules can be seen to define Huet's algorithm for (first-order) unification [Hue76]. This proceeds in two stages: in the first stage we saturate the equalities, and once they are saturated we perform the occurs-check. If implemented correctly, this will have complexity  $O(n \log(n))$ , where  $n$  is the size of the input problem. Robinson's original unification algorithm [Rob71] in contrast was exponential in the size of input, although in the context of his applications it performed quite well [CB83].

## 7 From Propositions to Rules of Inference

As we have seen in this lecture and also already in the last lecture, we can translate inference rules to propositions and then use either forward or backward chaining to specify and operational semantics for proof search.

Question: can we go the other way? That is, can we take *proposition* and turn them into inference rules? Another way to pose the question: can we take advantage of the chaining semantics to *compile* program propositions into “big-step” inference rules so we don't have to play through focusing all the time?

Let's try with the example of the type of the S combinator:

$$(a \supset (b \supset c)) \supset ((a \supset b) \supset (a \supset c))$$

In order to prove this, we first apply inversion as far as we can and arrive at the sequent

$$a, a \supset b, a \supset (b \supset c) \longrightarrow c$$

Now we have to decide on a polarization. We'll try all first all negative and then all positive.

Polarizing all atoms as negative we then have

$$a^-, \downarrow a^- \supset b^-, \downarrow a^- \supset (\downarrow b^- \supset c^-) \xrightarrow{f} c^-$$

Which propositions could we focus on? Not on  $c^-$  in the succedent—there is no rule for that in the backward chaining fragment. But we can focus on each of the antecedents since they are all negative propositions. In each case we imagine what would happen if we focused on the proposition, not knowing the remaining antecedents  $\Gamma^-$  or the conclusion  $P^-$ . But note that since antecedents are persistent, all propositions in  $\Gamma_0 = (a^-, \downarrow a^- \supset b^-, \downarrow a^- \supset (\downarrow b^- \supset c^-))$  will always be present in  $\Gamma$ .

$$\frac{\frac{\frac{a^- = P^-}{\Gamma^-, [a^-] \xrightarrow{f} P^-} \text{id}^-}{\Gamma^-, \downarrow a^- \supset P^-} \text{focus}L}{\Gamma^- \xrightarrow{f} P^-}$$

So for the focus on  $a^-$  to succeed, the right-hand side  $P^-$  must be equal to  $a^-$ . This gives use the derived rule

$$\frac{}{\Gamma^- \xrightarrow{f} a^-} R_1$$

Focusing on the second proposition:

$$\frac{\frac{\frac{P^- = b^-}{\Gamma^-, [b^-] \xrightarrow{f} P^-} \text{id}^- \quad \frac{\frac{\Gamma^- \xrightarrow{f} a^-}{\Gamma^- \xrightarrow{f} [\downarrow a^-]} \downarrow R}{\Gamma^-, [\downarrow a^- \supset b^-] \xrightarrow{f} P^-} \supset L}{\Gamma^-, [\downarrow a^- \supset b^-] \xrightarrow{f} P^-} \text{focus}L}{\Gamma^- \xrightarrow{f} P^-}$$

We see  $P^- = b^-$  and we have one stable subgoal that will be the premise of the derived rule.

$$\frac{\Gamma^- \xrightarrow{f} a^-}{\Gamma^- \xrightarrow{f} b^-} R_2$$

Finally, focusing on the third antecedent:

$$\frac{\frac{\frac{c^- = P^-}{\Gamma^-, [c^-] \xrightarrow{f} P^-} \text{id}^- \quad \frac{\Gamma^- \xrightarrow{f} b^-}{\Gamma^- \xrightarrow{f} [\downarrow b^-]} \downarrow R}{\Gamma^-, [\downarrow b^- \supset c^-] \xrightarrow{f} P^-} \supset L \quad \frac{\Gamma^- \xrightarrow{f} a^-}{\Gamma^- \xrightarrow{f} [\downarrow a^-]} \downarrow R}{\Gamma^-, [\downarrow a^- \supset (\downarrow b^- \supset c^-)] \xrightarrow{f} P^-} \supset L}{\Gamma^- \xrightarrow{f} P^-} \text{focus}_L$$

We read off  $P^- = c^-$  and the two stable sequents that are the premises of the derived rules:

$$\frac{\Gamma^- \xrightarrow{f} b^- \quad \Gamma^- \xrightarrow{f} a^-}{\Gamma^- \xrightarrow{f} c^-} R_3$$

Summarizing the three rules:

$$\frac{}{\Gamma^- \xrightarrow{f} a^-} R_1 \quad \frac{\Gamma^- \xrightarrow{f} a^-}{\Gamma^- \xrightarrow{f} b^-} R_2 \quad \frac{\Gamma^- \xrightarrow{f} b^- \quad \Gamma^- \xrightarrow{f} a^-}{\Gamma^- \xrightarrow{f} c^-} R_3$$

To see how they prove our original sequent

$$a^-, \downarrow a^- \supset b^-, \downarrow a^- \supset (\downarrow b^- \supset c^-) \xrightarrow{f} c^-$$

we first take all the negative propositions away from the antecedents. That's because *instead* of focusing on them, we should be using a derived rule. Here is the resulting proof

$$\frac{\frac{\frac{\frac{}{R_1}}{\cdot \xrightarrow{f} a^-} R_2 \quad \frac{}{R_1}}{\cdot \xrightarrow{f} b^-} R_3}{\cdot \xrightarrow{f} a^-} R_3}{\cdot \xrightarrow{f} c^-}$$

Of course, this is much shorter and more efficient than the proof using the explicit focusing rules.

Circling back: let's make all atoms positive

$$a^+, a^+ \supset \uparrow b^+, a^+ \supset (b^+ \supset \uparrow c^+) \xrightarrow{f} c^+$$

This time, we cannot focus on  $a^+$  in the antecedent, but on the other two propositions and also on the succedent. Let's do each in turn. Again, remember that  $\Gamma_0 = (a^+ \supset \uparrow b^+, a^+ \supset (b^+ \supset \uparrow c^+))$  is a part of every sequent in a proof. The succedent can be any positive proposition  $G^+$ .

$$\frac{\frac{\frac{a^+ \in \Gamma}{\Gamma \xrightarrow{f} [a^+]}}{\text{id}^+} \quad \frac{\Gamma, b^+ \xrightarrow{f} G^+}{\Gamma, [\uparrow b^+] \xrightarrow{f} G^+} \uparrow L}{\Gamma, [a^+ \supset \uparrow b^+] \xrightarrow{f} G^+} \supset L$$

$$\frac{\Gamma, [a^+ \supset \uparrow b^+] \xrightarrow{f} G^+}{\Gamma \xrightarrow{f} G^+} \text{focus} L$$

Here,  $a^+ \in \Gamma$  should be seen as a constraint, so we just write  $\Gamma = (\Gamma', a^+)$  and obtain the rule

$$\frac{\Gamma', a^+, b^+ \xrightarrow{f} G^+}{\Gamma', a^+ \xrightarrow{f} G^+} S_1$$

For the second proposition:

$$\frac{\frac{\frac{b^+ \in \Gamma}{\Gamma \xrightarrow{f} [b^+]}}{\text{id}^+} \quad \frac{\frac{\Gamma, c^+ \xrightarrow{f} G^+}{\Gamma, [\uparrow]c^+ \xrightarrow{f} G^+} \uparrow L}{\Gamma, [b^+ \supset \uparrow c^+] \xrightarrow{f} G^+} \supset L}{\Gamma, [a^+ \supset (b^+ \supset \uparrow c^+)] \xrightarrow{f} G^+} \supset L$$

$$\frac{\Gamma, [a^+ \supset (b^+ \supset \uparrow c^+)] \xrightarrow{f} G^+}{\Gamma \xrightarrow{f} G^+} \text{focus} L$$

Again, collecting membership constraints we see  $\Gamma = (\Gamma', a^+, b^+)$  and we get

$$\frac{\Gamma', a^+, b^+, c^+ \xrightarrow{f} G^+}{\Gamma', a^+, b^+ \xrightarrow{f} G^+} S_2$$

Finally, we can focus on the succedent:

$$\frac{\frac{\frac{c^+ \in \Gamma}{\Gamma \xrightarrow{f} [c^+]}}{\Gamma \xrightarrow{f} c^+} \text{id}^+}{\text{focus}R}$$

which gives us

$$\frac{}{\Gamma', c^+ \xrightarrow{f} c^+} S_3$$

In summary:

$$\frac{\Gamma', a^+, b^+ \xrightarrow{f} G^+}{\Gamma', a^+ \xrightarrow{f} G^+} S_1 \quad \frac{\Gamma', a^+, b^+, c^+ \xrightarrow{f} G^+}{\Gamma', a^+, b^+ \xrightarrow{f} G^+} S_2 \quad \frac{}{\Gamma', c^+ \xrightarrow{f} c^+} S_3$$

With these three rules we can drop  $\Gamma_0$  since their only purpose would be to focus on them—and that has been replaced by the derived rules. Our big-step proof becomes:

$$\frac{\frac{\frac{a^+, b^+, c^+ \xrightarrow{f} c^+}{a^+, b^+ \xrightarrow{f} c^+} S_2}{a^+ \xrightarrow{f} c^+} S_1}{a^+ \xrightarrow{f} c^+} S_3$$

Note that we have kept  $a^+$  among the antecedents since we cannot focus on a positive atom (or any positive proposition, for that matter) in the antecedent.

## 8 Polarization: A Brief Roadmap

A critical component in understanding the various fragment and operational interpretations we have seen is *polarization* [Lau99]. We started from the inversion strategy.

**Negative Propositions** are those with invertible right rules, that is, if we see them in the succedent we can apply their right rule without considering any other choice and search remains complete.

**Positive Propositions** are those with invertible left rules, that is, if we see them in the antecedent we can apply their left rule without considering any other choice and search remains complete.

In order for *every* proposition to have a polarized version we need the so-called *shift* operators  $\uparrow$  and  $\downarrow$  to go between the two classes of propositions. We get:

$$\begin{aligned} \text{Neg. props. } A^- &::= P^- \mid B^+ \supset A^- \mid \forall x. A^-(x) \mid A_1^- \wedge A_2^- \mid \top \mid \uparrow B^+ \\ \text{Pos. props. } B^+ &::= P^+ \mid B_1^+ \wedge B_2^+ \mid \top \mid \exists x. B^+(x) \mid B_1^+ \vee B_2^+ \mid \perp \mid \downarrow A^- \end{aligned}$$

Atoms can be either negative ( $P^-$ ) or positive ( $P^+$ ). During polarization we can choose the polarization of each atom  $p$  freely, but must assign the same polarity to each occurrence of  $p$ . Conjunction and truth have invertible left and right rules, so they appear in both rows.

*Chaining* is the opposite of inversion: we focus on a particular negative antecedent or positive succedent and continue to apply rules only to the single proposition in focus until the focusing phase is interrupted by a shift, changing the polarity of the proposition.

Chaining by itself is complete for proof search as long as the shifts are restricted such that we only have  $B^- ::= \dots \mid \downarrow P^-$  and  $A^- ::= \dots \mid \uparrow P^+$ . The language so restricted is (an insignificant extension) of Horn logic.

In case all atoms are negative, chaining is called *backward chaining* (also called *top-down logic programming*), which is a goal-directed proof search strategy and the foundation of Prolog [Kow88].

In case all atoms are positive, chaining is called *forward chaining* (also called *bottom-up logic programming* [NR91]), which is a saturation-based proof search strategy and the foundation of Datalog.

There is the possibility of allowing both positive and negative atoms, but the resulting mixed chaining logic programming language has never been deeply investigated, as far as I am aware. However, chaining is complete for this language, so it is a plausible candidate for an interesting and expressive language.

If we allow arbitrary polarized propositions, then chaining alone is insufficient: positive (non-atomic) propositions can show up in the antecedents, and negative (non-atomic) propositions in the succedent. When such a proposition is encountered, we apply inversion until we once again reach a *stable sequent* which is characterized with only negative propositions and positive atoms as antecedents, and positive propositions and negative atoms as succedents.

*Focusing = chaining + inversion* was first discovered by Andreoli [And92], with two caveats: (1) his propositions were not implicitly polarized, and (2) focusing was defined for *linear logic* [Gir87], which we will only see later in this course. However, focusing (and also the chaining-only fragment based on Horn logic) has been remarkably robust in that it applies to a large number of reasonable substructural, modal, and other logics, both intuitionistic and classical.

We will discuss focusing in the next lecture.

## References

- [And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.
- [CB83] Jacques Corbin and Michel Bidoit. Rehabilitation of Robinson’s unification algorithm. In *Information Processing 83*, volume 9, pages 909–914, 1983.
- [Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [Hue76] Gérard Huet. *Résolution d’équations dans des langages d’ordre 1, 2, . . . , ω*. PhD thesis, Université Paris VII, September 1976.
- [Kow88] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.
- [Lau99] Olivier Laurent. Polarized proof-nets: Proof-nets for LC. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA 1999)*, pages 213–227, L’Aquila, Italy, April 1999. Springer LNCS 1581.
- [NR91] Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in Honor of Alan Robinson*, pages 640–700. MIT Press, Cambridge, Massachusetts, 1991.
- [Rob71] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

# Lecture Notes on Focusing

15-317: Constructive Logic  
Frank Pfenning

Lecture 19  
November 7, 2017

## 1 Introduction

In this lecture we will finally put much of what we have learned on proof theory together, following the slogan *focusing = inversion + chaining*. Focusing has been developed by Andreoli [And92] using classical linear logic, but it has proved to be a remarkably robust concept (see, for example, Liang and Miller [LM09]). We will follow the formulation of Simmons [Sim14], which includes particularly elegant proofs of the completeness of focusing using structural inductions.

## 2 Polarization

A key idea behind focusing is to limit nondeterminism by sequencing inferences on connectives that have similar behaviors. One behavior is that of *inversion*, perhaps slightly misnamed. Andreoli calls such connectives *asynchronous*, which expresses that when we see such a connective we can always decompose it. *Synchronous* connectives, by contrast, are those that “may have to wait” until they can be decomposed, but once we have committed to one by *focusing* on it, we can continue to chain inferences on this one proposition and don’t need to look elsewhere.

These concepts match perfectly in the sense that a connective that is *asynchronous* when it appears as a succedent will be *synchronous* as an antecedent. Intuitively, this derives from the nature of *harmony* between the right and the left rules as witnessed by cut reduction. A rule inferring and

asynchronous proposition carries no information (the premise and conclusion are interderivable and therefore the rules does not gain or loose information), while a rule inferring a synchronous proposition has to make a choice of some form. This choice is information which is “conveyed” to the asynchronous connective.

If we classify propositions by their behavior as *succedents*, then so-called *negative propositions* are asynchronous or, to say it differently, have invertible right rules. Conversely, *positive propositions* are asynchronous when they appear as *antecedents*, or, to say it differently, have invertible left rules. The so-called *shift operators* go back and forth between positive and negative propositions so that any proposition can be polarized. n

$$\begin{array}{ll} \text{Neg. Props. } A^-, B^- & ::= A^+ \supset B^- \mid A^- \wedge B^- \mid \top \mid P^- \mid \uparrow A^+ \\ \text{Pos. Props. } A^+, B^+ & ::= A^+ \vee B^+ \mid \perp \mid A^+ \wedge B^+ \mid \top \mid P^+ \mid \downarrow A^- \end{array}$$

A few notes:

**Conjunction and truth:** Conjunction  $A \wedge B$  and truth  $\top$  appear as both positive and negative propositions. That’s because There are invertible rules for conjunction both in the antecedent and the succedent. Really, it should be seen as an indication that there are two different conjunctions  $A^- \wedge^- B^-$  and  $A^+ \wedge^+ B^+$  and two different truth constants  $\top^-$  and  $\top^+$  with different rules that happen to be *logically equivalent* even though they have different intrinsic properties, both from the perspective of proof search and the computational contents of proofs. For example, in a functional language, positive conjunction would correspond to an eager pairs, while negative conjunction corresponds to lazy pairs.

So, if we take proofs seriously as defining the meaning of propositions there should be two conjunctions, which are disambiguated in the polarized presentation of logic.

**Atoms:** Atoms may be viewed from one perspective as propositional variables, from another as “uninterpreted” propositions which means that only the logical assumptions we make about them imbue them with meaning. Each can be independently assiged an arbitrary polarity, as long as all occurrences of an atom are given the same polarity.

**Quantifiers:** The universal quantifier is negative since its right rule is invertible, while the existential quantifier is positive. We do not treat them formally to avoid the syntactic complication of introducing terms, parameters, their types, and the relevant typing judgments.

### 3 Inversion

Inversion decomposes all asynchronous connectives until we reach a sequent where all proposition in the sequent are either atoms or synchronous. In order for inversion to proceed deterministically, first decompose asynchronous connectives in the succedent and then in the antecedent. We use an *ordered context*  $\Omega^+$  (as in [Lecture 12](#)) consisting of all positive propositions.

$$\begin{array}{ll} \text{Stable succedent} & \rho ::= A^+ \mid P^- \\ \text{Stable antecedents} & \Gamma ::= \cdot \mid \Gamma, A^- \mid \Gamma, P^+ \\ \text{Right inversion} & \Gamma ; \Omega^+ \xrightarrow{R} A^- \\ \text{Left inversion} & \Gamma ; \Omega^+ \xrightarrow{L} \rho \\ \text{Stable sequent} & \Gamma ; \rho \end{array}$$

The rules are summarized in Figure 1

### 4 Chaining

Once inversion has completed, we have to focus on a single proposition, either a positive succedent or a negative antecedents, and then chain together inference on the proposition in focus. In particular, no other propositions are considered, and only one proposition can be in focus. This gives us two new forms of judgments.

$$\begin{array}{ll} \text{Right focus} & \Gamma \longrightarrow [A^+] \\ \text{Left focus} & \Gamma, [A^-] \longrightarrow \rho \end{array}$$

The rules can be found in Figure 2. Some remarks:

**Atoms:** Much of the power of focusing comes from the fact that left focus  $[P^-]$  fails unless the succedent is also  $P^-$ . Dually, right focus  $[P^+]$  fails unless  $P^+$  is one of the antecedents. Note also that it is not possible to focus on a positive atom in the antecedent or a negative atom in the succedent.

**Shifts:** In contrast,  $\uparrow L$  and  $\downarrow R$  just lose focus and return to the appropriate inversion judgment.

$$\begin{array}{c}
\frac{\Gamma ; \Omega^+ \cdot A^+ \xrightarrow{R} B^-}{\Gamma ; \Omega^+ \xrightarrow{R} A^+ \supset B^-} \supset R \\
\\
\frac{\Gamma ; \Omega^+ \xrightarrow{R} A^- \quad \Gamma ; \Omega^+ \xrightarrow{R} B^-}{\Gamma ; \Omega^+ \xrightarrow{R} A^- \wedge B^-} \wedge R \qquad \frac{}{\Gamma ; \Omega^+ \xrightarrow{R} \top} \top R \\
\\
\frac{\Gamma ; \Omega^+ \xrightarrow{L} P^-}{\Gamma ; \Omega^+ \xrightarrow{R} P^-} pR \qquad \frac{\Gamma ; \Omega^+ \xrightarrow{L} A^+}{\Gamma ; \Omega^+ \xrightarrow{R} \downarrow A^+} \uparrow R \\
\\
\frac{\Gamma ; \Omega^+ \cdot A^+ \xrightarrow{L} \rho \quad \Gamma ; \Omega^+ \cdot B^+ \xrightarrow{L} \rho}{\Gamma ; \Omega^+ \cdot A^+ \vee B^+ \xrightarrow{L} \rho} \vee L \qquad \frac{}{\Gamma ; \Omega^+ \cdot \perp \xrightarrow{L} \rho} \perp L \\
\\
\frac{\Gamma ; \Omega^+ \cdot A^+ \cdot B^+ \xrightarrow{L} \rho}{\Gamma ; \Omega^+ \cdot A^+ \wedge B^+ \xrightarrow{L} \rho} \wedge L \qquad \frac{\Gamma ; \Omega^+ \xrightarrow{L} \rho}{\Gamma ; \Omega^+ \cdot \top \xrightarrow{L} \rho} \top L \\
\\
\frac{\Gamma, P^+ ; \Omega^+ \xrightarrow{L} \rho}{\Gamma ; \Omega^+ \cdot P^+ \xrightarrow{L} \rho} pL \qquad \frac{\Gamma, A^- ; \Omega^+ \xrightarrow{L} \rho}{\Gamma ; \Omega^+ \cdot \downarrow A^- \xrightarrow{L} \rho} \downarrow L \\
\\
\frac{\Gamma \longrightarrow \rho}{\Gamma ; \cdot \xrightarrow{L} \rho} \text{stable}
\end{array}$$

Figure 1: Inversion phase of focusing

$$\begin{array}{c}
\frac{\Gamma \longrightarrow [A^+]}{\Gamma \longrightarrow A^+} \text{ focus } R \quad \frac{\Gamma, [A^-] \longrightarrow \rho}{\Gamma, A^- \longrightarrow \rho} \text{ focus } L \\
\\
\frac{\Gamma \longrightarrow [A^+]}{\Gamma \longrightarrow [A^+ \vee B^+]} \vee R_1 \quad \frac{\Gamma \longrightarrow [B^+]}{\Gamma \longrightarrow [A^+ \vee B^+]} \vee R_2 \quad \text{no right rule for } [\perp] \\
\\
\frac{\Gamma \longrightarrow [A^+] \quad \Gamma \longrightarrow [B^+]}{\Gamma \longrightarrow [A^+ \wedge B^+]} \wedge R \quad \frac{}{\Gamma \longrightarrow [\top]} \top R \\
\\
\frac{}{\Gamma, P^+ \longrightarrow [P^+]} \text{id}^+ \quad \frac{\Gamma ; \cdot \xrightarrow{R} A^-}{\Gamma \longrightarrow [\downarrow A^-]} \downarrow R \\
\\
\frac{\Gamma \longrightarrow [A^+] \quad \Gamma, [B^-] \longrightarrow \rho}{\Gamma, [A^+ \supset B^-] \longrightarrow \rho} \supset L \\
\\
\frac{\Gamma, [A^-] \longrightarrow \rho \quad \Gamma, [B^-] \longrightarrow \rho}{\Gamma, [A^- \wedge B^-] \longrightarrow \rho} \wedge L_1 \quad \frac{\Gamma, [B^-] \longrightarrow \rho}{\Gamma, [A^- \wedge B^-] \longrightarrow \rho} \wedge L_2 \quad \text{no left rule for } [\top] \\
\\
\frac{}{\Gamma, [P^-] \longrightarrow P^-} \text{id}^- \quad \frac{\Gamma ; A^+ \xrightarrow{L} \rho}{\Gamma, [\uparrow A^+] \longrightarrow \rho} \uparrow L
\end{array}$$

Figure 2: Chaining phase of focusing

## 5 Deriving Rules, Revisited

In this more general setting when compared to chaining, deriving inference rules is slightly more complex: once the chaining phase completes, we have to complete the subsequent inversion phase until we arrive once again at stable sequents. We show a simple example, for atoms  $a$ ,  $b$ , and  $c$ .

$$a \wedge (a \supset (b \vee c)) \wedge (b \supset c) \supset c$$

First, we polarize the atoms. It looks as if  $a$  should be naturally positive (occurs only on the left-hand side of an implication or conjunction), while  $b$  and  $c$  are ambiguous. Let's make  $b$  positive and  $c$  negative. Then we polarize by inserting the minimal number of shifts.

$$a^+ \wedge \downarrow(a^+ \supset \uparrow(b^+ \vee \downarrow c^-)) \wedge (b^+ \supset c^-) \supset c^-$$

Overall, we have a negative proposition we start with

$$\cdot ; \cdot \xrightarrow{R} a^+ \wedge \downarrow(a^+ \supset \uparrow(b^+ \vee \downarrow c^-)) \wedge (b^+ \supset c^-) \supset c^-$$

and apply inversion until we reach a stable sequent, namely

$$a^+, a^+ \supset \uparrow(b^+ \vee \downarrow c^-), b^+ \supset c^- \longrightarrow c^-$$

We can only focus on the second and third antecedent. We derive:

$$\begin{array}{c} \vdots \\ \frac{\frac{\frac{a^+ \in \Gamma}{\Gamma \longrightarrow [a^+]} \text{id}^+ R \quad \frac{\Gamma ; b^+ \vee \downarrow c^- \xrightarrow{L} \rho}{\Gamma, [\uparrow(b^+ \vee \downarrow c^-)] \longrightarrow \rho} \uparrow L}{\Gamma, [a^+ \supset \uparrow(b^+ \vee \downarrow c^-)] \longrightarrow \rho} \supset L}{\Gamma, [a^+ \supset \uparrow(b^+ \vee \downarrow c^-)] \longrightarrow \rho} \end{array}$$

We see that when we lost focus due to the shift we switched over to a left inversion phase which we now complete.

$$\begin{array}{c} \frac{\frac{\frac{\Gamma, b^+ \longrightarrow \rho \text{ stable}}{\Gamma, b^+ ; \cdot \xrightarrow{L} \rho} pL \quad \frac{\frac{\Gamma, c^- \longrightarrow \rho \text{ stable}}{\Gamma, c^- ; \cdot \xrightarrow{L} \rho} \downarrow L}{\Gamma ; b^+ \xrightarrow{L} \rho \quad \Gamma ; \downarrow c^- \xrightarrow{L} \rho} \vee L}{\Gamma, [b^+ \supset \uparrow(b^+ \vee \downarrow c^-)] \longrightarrow \rho} \supset R \\ \frac{\frac{a^+ \in \Gamma}{\Gamma \longrightarrow [a^+]} \text{id}^+ R \quad \frac{\Gamma ; b^+ \vee \downarrow c^- \xrightarrow{L} \rho}{\Gamma, [\uparrow(b^+ \vee \downarrow c^-)] \longrightarrow \rho} \uparrow L}{\Gamma, [a^+ \supset \uparrow(b^+ \vee \downarrow c^-)] \longrightarrow \rho} \end{array}$$

Summarizing this rule, we obtain

$$\frac{\Gamma, a^+, b^+ \rightarrow \rho \quad \Gamma, a^+, c^- \rightarrow \rho}{\Gamma, a^+ \rightarrow \rho} R_1$$

This rules adds a negative atom  $c^-$  to the antecedents, so we need to derive another rule for it.

$$\frac{\frac{\rho = c^-}{\Gamma, c^-, [c^-] \rightarrow \rho} \text{id}^-}{\Gamma, c^- \rightarrow \rho} \text{focus}_L \quad \text{as a derived rule : } \frac{}{\Gamma, c^- \rightarrow c^-} R_2$$

And finally our original second antecedent:

$$\frac{\frac{\frac{b^+ \in \Gamma}{\Gamma \rightarrow [b^+]} \text{id}^+ \quad \frac{\rho = c^-}{\Gamma, [c^-] \rightarrow \rho} \text{id}^-}{\Gamma, [b^+ \supset c^-] \rightarrow \rho} \supset L}{\Gamma, b^+ \rightarrow c^-} R_3$$

Here is the summary of the three derived rules:

$$\frac{\Gamma, a^+, b^+ \rightarrow \rho \quad \Gamma, a^+, c^- \rightarrow \rho}{\Gamma, a^+ \rightarrow \rho} R_1 \quad \frac{}{\Gamma, c^- \rightarrow c^-} R_2 \quad \frac{}{\Gamma, b^+ \rightarrow c^-} R_3$$

This antecedents are persistent, we *replace* the two propositions which yielded  $R_1$  and  $R_3$  with the rules and we have to prove

$$a^+ \rightarrow c^-$$

which works as follows (where we are now only allowed to use derived rules):

$$\frac{\frac{}{a^+, b^+ \rightarrow c^-} R_3 \quad \frac{}{a^+, c^- \rightarrow c^-} R_2}{a^+ \rightarrow c^-} R_1$$

In this technique of deriving rules, each derived rules will only have stable sequents in the conclusion and premises. The rule generation will start with a negative antecedent or positive succedent, break it down until it encounters an atom, or an up or down shift, respectively, then proceed by inversion until another stable sequent is reached. Andreoli called propositions of this form *bipoles* because they traverse *negative* to *positive* or *positive* to *negative*, and back [And01].

## References

- [And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.
- [And01] Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1–3):131–163, 2001.
- [LM09] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, November 2009.
- [Sim14] Robert J. Simmons. Structural focalization. *Transactions on Computational Logic*, 15(3):21:1–21:33, July 2014.

# Lecture Notes on Substructural Deduction

15-317: Constructive Logic  
Jonathan Sterling\*

Lecture 20  
November 13, 2017

In philosophy, the *subjective logic* is the study of valid inference: the correct deduction of conclusions from premises. We transcribe this activity using horizontal lines, where premises lie above the line and conclusions lie below the line:

$$\frac{\mathcal{J}_0 \quad \cdots \quad \mathcal{J}_n}{\mathcal{K}} R$$

The meaning of such an inference is that we can deduce  $\mathcal{K}$  from  $\mathcal{J}_0$  through  $\mathcal{J}_n$ . This is a schema for the deduction of *mathematical* knowledge, which remains true in permanence as soon as it has become true. Therefore, deductions of this kind may freely ignore, reuse or reorder known facts; this kind of inference is called *structural*, and notions of inference which are not closed under these operations are called *substructural*.

## 1 Linear Inference

What we have seen above is not, however, the full picture, which is captured instead by the notion of inference qua *local transformation* of states. In generality, such a view requires us to admit multiple conclusions, leading to the following schema for inference:

$$\frac{\mathcal{J}_0 \quad \cdots \quad \mathcal{J}_m}{\mathcal{K}_0 \quad \cdots \quad \mathcal{K}_n} R$$

---

\*These notes are substantially influenced by Frank Pfenning's notes on Deductive Inference from 15-816 (2016).

The meaning of the above inference is that the transition  $R$  can take place when our state includes  $\mathcal{J}_0$  through  $\mathcal{J}_m$ , and takes us to a new state which *instead* has  $\mathcal{K}_0$  through  $\mathcal{K}_n$ . In this transition, the facts  $\mathcal{J}_i$  have been consumed and replaced.

Under such a paradigm, a rule of inference expresses an atomic transition from one state to another; the premises capture a fragment of the state prior to the transition, and the conclusions capture the effect of that transition on the original state.

**Ephemerality and persistence** The above explanation captures the more general situation of knowledge which is not *persistent* but instead *ephemeral* (subject to loss or change). We call inference that involves ephemeral judgments *linear*.

The ephemeral version of deduction which we described above can be used to describe everyday situations, where the subjects of inference are not unchanging realities but are instead contingent and subject to change over time.

While the status of a statement like  $\forall a, b, c : \mathbb{R}. a^2 + b^2 = c^2$  is permanent or persistent from the moment of its deduction, consider the statement "*China is on the socialist road*". This assertion may be correct at one time and incorrect at another; it began to be true in 1949, but it ceased to be true in 1976. *Truth is ephemeral*.

## 1.1 A Single Spark Can Start A Prairie Fire

Consider the old Chinese saying, "*A single spark can start a prairie fire*." We can capture this idea using linear inference using the following kinds of fact: *spark*, *on\_fire(p)* and *prairie(p)*.

$$\frac{\text{prairie}(p) \quad \text{spark}}{\text{prairie}(p) \quad \text{on\_fire}(p)}$$

We can refine this a bit more by noticing that some  $p$  being a prairie is a *persistent* kind of fact: once true, it always remains true, and we can duplicate that fact as-needed. We will use underlines to indicate persistent facts in our states; we are also justified in omitting a persistent fact in the conclusion, since it is implicitly present.

$$\frac{\underline{\text{prairie}(p)} \quad \text{spark}}{\text{on\_fire}(p)}$$

Choosing substructural (linear, ordered) deduction is not a restriction so much as an ability. *Substructural logic generalizes structural logic.*

## 1.2 Commodity Exchange, the Universal Equivalent and the Money-Form of Value

In political economy, the *Value form* is initially comprised of two parts: the *relative form of value* and the *equivalent form of value* [MMF04]. These two moments of value can be observed when two commodities meet in the marketplace for exchange; for instance:

20 yards of linen  
one coat

If a quantity of coats is fixed, then the relative form of value is the 20 yards of linen, and the equivalent form is the single coat; when the exchange is read in the other direction (and the quantity of linen is fixed), the relative form is the single coat and the equivalent form is the 20 yards of linen.

Conceivably one can come up with relative and equivalent values for all commodities, leading to a vast network of relationships between commodities; for instance, if one can establish an exchange rate between coats and pomegranates, then one can compose this with the one between linen and coats, establishing a certain quantity of pomegranates as the equivalent form of value to the relative form of value in linen.

However, the development of an advanced capitalist economy is essentially precluded by this design, because of the fact that the values of each commodity depend on the values of every other commodity that they can (transitively) be exchanged with. There are two main problems:

1. Whether one can actually exchange commodity *A* with commodity *B* is contingent on whether there is a chain of exchange relations that connects *A* and *B*; this is not guaranteed.
2. The units of exchange between two commodities may establish a non-whole-number relative value at a certain equivalent value. This can lead to unfair exchanges.

The answer to these problems is the development of a *universal equivalent* against which the relative form of value of all commodities is measured, i.e. a *classifying object* in the space of commodities. This universal equivalent is called **money**. When we have a universal equivalent, the simplest way to exchange two commodities is to use money as an intermediate form.

$$\begin{array}{c}
 \text{pretzel} & \text{bagel} & \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} \\
 \hline
 \text{quarter} & \text{dollar} & \text{quarter} & & & \\
 & & & & \text{dollar} &
 \end{array}$$

Now, suppose I have seven pretzels, and I want to convert these into bagels. Let us bring the pretzels to the market:

$$\begin{array}{cccccccc}
 \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} \\
 \hline
 \text{quarter} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} \\
 \hline
 \text{quarter} & \text{quarter} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} \\
 \hline
 \text{quarter} & \text{quarter} & \text{quarter} & \text{pretzel} & \text{pretzel} & \text{pretzel} & \text{pretzel} \\
 \hline
 \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{pretzel} & \text{pretzel} & \text{pretzel} \\
 \hline
 \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{pretzel} & \text{pretzel} \\
 \hline
 \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{pretzel} \\
 \hline
 \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} & \text{quarter} \\
 \hline
 \text{dollar} & \text{quarter} & \text{quarter} & \text{quarter} & & & \\
 \hline
 \text{bagel} & \text{quarter} & \text{quarter} & & & &
 \end{array}$$

Using the rules of inference, we first converted all of our pretzels into the universal equivalent; then, we converted this into as many bagels as possible. At the end, we received 50¢ in change from the bagel vendor, because our pretzels did not have the relative form of value of a whole-number equivalent quantity of bagel.

## 2 Ordered Inference

So far we have uncovered a notion of inference which describes the local transition of states, which are comprised of unordered collections of facts. We can refine this further, by allowing these facts to have a *location* relative to one another, and not generally allowing two facts to switch places. This opens up an entirely new realm of processes which can be captured through inference.

## 2.1 Example: A Stack Calculator

Ordered inference can be used to encode what is called *substructural operational semantics*, in which transitions for state machines are encoded as rules of inference [PS09]; the ordered character of facts naturally gives rise to a notion of control stack.

Consider the following following grammar of arithmetic expressions and stack frames:

$$\begin{array}{lcl} E & ::= & \bar{n} \mid \text{plus}(E, E) \mid \text{minus}(E, E) \\ K & ::= & \text{plus}(\square, E) \mid \text{plus}(\bar{n}, \square) \mid \text{minus}(\square, E) \mid \text{minus}(\bar{n}, \square) \end{array}$$

Now we define three kinds of fact or task:

1.  $\text{calc}(E)$  means “calculate  $E$ ”
2.  $\text{ret}(\bar{n})$  means “return the numeral  $\bar{n}$ ”
3.  $\text{cont}(K)$  means “resume calculation  $K$  when a value has been returned”

We now provide rules which, if executed in *ordered* inference, will derive  $\text{ret}(\bar{n})$  for some  $\bar{n}$  from  $\text{calc}(E)$  for any  $E$ .

$$\begin{array}{c} \text{calc}(\bar{n}) \qquad \frac{\text{calc}(\text{plus}(E_1, E_2))}{\text{calc}(E_1) \quad \text{cont}(\text{plus}(\square, E_2))} \qquad \frac{\text{calc}(\text{minus}(E_1, E_2))}{\text{calc}(E_1) \quad \text{cont}(\text{minus}(\square, E_2))} \\ \hline \text{ret}(\bar{n}) \qquad \qquad \qquad \qquad \qquad \qquad \qquad \end{array}$$

$$\begin{array}{c} \frac{\text{ret}(\bar{n}) \quad \text{cont}(\text{plus}(\square, E))}{\text{calc}(E) \quad \text{cont}(\text{plus}(\bar{n}, \square))} \qquad \frac{\text{ret}(\bar{n}) \quad \text{cont}(\text{plus}(\bar{m}, \square))}{\text{ret}(\bar{n} + m)} \\ \hline \qquad \qquad \qquad \qquad \qquad \qquad \qquad \end{array}$$

$$\begin{array}{c} \frac{\text{ret}(\bar{n}) \quad \text{cont}(\text{minus}(\square, E))}{\text{calc}(E) \quad \text{cont}(\text{minus}(\bar{n}, \square))} \qquad \frac{\text{ret}(\bar{n}) \quad \text{cont}(\text{minus}(\bar{m}, \square))}{\text{ret}(\bar{n} - m)} \\ \hline \qquad \qquad \qquad \qquad \qquad \qquad \qquad \end{array}$$

Observe that the *ordered* character of inference is crucial here. If we were allowed to reorder facts, the procedure would be non-deterministic and would in most cases not return the correct result: consider what would happen if we swapped two different stack frames!

**Exercise 1** In order to encode our control stack, we have exploited the fact that facts cannot be exchanged in ordered inference. Suppose we were working in linear inference, where facts can be exchanged; can you still find a way to encode the stack calculator?

## 2.2 Lambek Calculus

The first application of ordered deduction was to capture the surface structure of natural language expressions in a compositional way, accounting for all the oddities of word order, gapping, and scrambling which pervade human languages. Initiated by Lambek [Lam58], this tradition has given birth to an explosion of new calculi and formalisms for ordered logic during the past half century, both in the style of lambda calculus [PP98, PP, MVF11, Mor12] and in combinatory style [Ste96, Bal02].

Lambek calculus begins by specifying a collection “atomic syntactic types” (base types) which represent the primitive parts of speech. Here are some basic ones:<sup>1</sup>

TYPE	MEANING	EXAMPLE
np	noun phrase	“people”
dp	determiner phrase	“the people”, “she”
vp	verb phrase	“she serves the people”

Then, we add two connectives  $B / A$  and  $A \setminus B$  (called “over” and “under” respectively) which are subject to the following rules of ordered inference:

$$\frac{M : B / A \quad N : A}{M N : B} /E \quad \frac{N : A \quad M : A \setminus B}{N M : B} \setminus E$$

From the elimination rules above, these seem to be some kind of implication, which differ only in whether the argument should occur to the left or to the right. We use these connectives to generate the types of words, and then the rules of the logic specify the ways that words can be combined.

For instance, transitive verbs have type  $(dp \setminus vp) / dp$ ; that is, they are terms which take (first) a determiner phrase on the right (the object) and then another determiner phrase on the left (the subject), and then behave as a verb phrase (roughly a sentence). Adjectives have type  $np / np$ : if you place one before a noun phrase, it becomes a noun phrase. Determiners have type  $dp / np$ : if you place a determiner before a noun phrase, it becomes a determiner phrase.

Conjunctions like “and” can be assigned the type  $A \setminus (A / A)$  for any syntactic type  $A$ .

---

<sup>1</sup>In modern linguistics, there are many more syntactic types! But these will allow us to work through some basic examples. Note also that in these notes we diverge slightly from the atomic types chosen by Lambek, preferring a presentation more aligned with the modern understanding of natural language syntax.

### 2.2.1 Parsing

With these connectives in place, ordered inference gives an operational semantics to parsing problems. To set up a parsing problem, the initial state of facts is given as a sequence of words together with their syntactic type; then, one tries to use the rules of Lambek calculus to derive a single proposition, vp. Here is an example:

$$\begin{array}{c}
 \frac{\text{Women : dp} \quad \text{hold up} : (\text{dp} \setminus \text{vp}) / \text{dp} \quad \text{half} : \text{dp} / \text{dp} \quad \text{the} : \text{dp} / \text{np} \quad \text{sky} : \text{np}}{\text{Women : dp} \quad \text{hold up} : (\text{dp} \setminus \text{vp}) / \text{dp} \quad \text{half} : \text{dp} / \text{dp} \quad \text{the sky} : \text{dp}} /E \\
 \frac{}{\text{Women : dp} \quad \text{hold up} : (\text{dp} \setminus \text{vp}) / \text{dp} \quad \text{half} : \text{dp} / \text{dp} \quad \text{the sky} : \text{dp}} /E \\
 \frac{}{\text{Women : dp} \quad \text{hold up} : (\text{dp} \setminus \text{vp}) / \text{dp} \quad \text{half the sky} : \text{dp}} /E \\
 \frac{}{\text{Women : dp} \quad \text{hold up half the sky} : \text{dp} \setminus \text{vp}} /E \\
 \frac{}{\text{Women hold up half the sky} : \text{vp}} \backslash E
 \end{array}$$

**Remark 1** *The reader may be uncomfortable with the fact that we have started with  $\text{hold up} : (\text{dp} \setminus \text{vp}) / \text{dp}$  as a single lexeme, rather than as a compound phrase formed using the verb  $\text{hold}$  and the verb modifier  $\text{up}$ . Unfortunately, it is not yet possible for us to account for this kind of gapping construction, which requires the machinery developed in [MVF11].*

**Remark 2** *Why is  $\text{women} : \text{dp}$  a determiner phrase rather than a noun phrase? Technically, it is really a plural noun phrase which is adjoined to a silent plural determiner; we will learn to account for phenomena analogous to this in §2.2.3.*

### 2.2.2 Derived Rules and Type Raising

For the sake of space, fix the following abbreviations:

$$\begin{array}{ll}
 \text{tv} \triangleq (\text{dp} \setminus \text{vp}) / \text{dp} & \text{(transitive verb)} \\
 \text{iv} \triangleq \text{dp} \setminus \text{vp} & \text{(intransitive verb)} \\
 \text{d} \triangleq \text{dp} / \text{np} & \text{(determiner)} \\
 \text{conj}[A] \triangleq A \setminus (A / A) & \text{(conjunction)}
 \end{array}$$

Try to parse the sentence, “*Lenin opposes and Kerensky supports the War.*”, assuming the following initial state, writing  $?$  for an unconstrained syntactic type:

$\text{Lenin} : \text{dp} \quad \text{opposes} : \text{tv} \quad \text{and} : \text{conj}[?] \quad \text{Kerensky} : \text{dp} \quad \text{supports} : \text{tv} \quad \text{the} : \text{d} \quad \text{war} : \text{np}$

You will quickly find that it cannot be done! Certain parts of the sentence can be parsed, but no matter what we try, we will get stuck. For instance:

$$\begin{array}{c}
 \frac{\text{Lenin : dp } \text{ opposes : tv } \text{ and : conj[?]} \text{ Kerensky : dp } \text{ supports : tv } \text{ the : d } \text{ war : np}}{\text{Lenin : dp } \text{ opposes : tv } \text{ and : conj[?]} \text{ Kerensky : dp } \text{ supports : tv } \text{ the war : dp}} /E \\
 \frac{\text{Lenin : dp } \text{ opposes : tv } \text{ and : conj[?]} \text{ Kerensky : dp } \text{ supports : tv } \text{ the war : dp}}{\text{Lenin : dp } \text{ opposes : tv } \text{ and : conj[?]} \text{ Kerensky : dp } \text{ supports the war : iv}} /E \\
 \frac{\text{Lenin : dp } \text{ opposes : tv } \text{ and : conj[?]} \text{ Kerensky : dp } \text{ supports the war : iv}}{\text{Lenin : dp } \text{ opposes : tv } \text{ and : conj[?]} \text{ Kerensky supports the war : vp}} \backslash E
 \end{array}$$

And that is as far as we can possibly get. To get further, we need to introduce *type raising*. First, let's add introduction rules for the *over* and *under* connectives:

$$\frac{\overline{\dots u : A} \quad \vdots}{M : B} /I^u \quad \frac{u : A \dots \quad \vdots}{M : B} \backslash I^u$$

In the above rules, we use ellipses to indicate that the hypothesis must be the rightmost or the leftmost assumption respectively. Now, for any syntactic types  $A, X$  we can derive the following *type raising* rule:

$$\frac{M : A}{M^\uparrow : X / (A \setminus X)} \text{ raise} \triangleq \frac{\frac{M : A \quad \overline{v : A \setminus X}}{v M : X} \backslash E}{\lambda v. v M : X / (A \setminus X)} /I^v$$

Another important derived rule is *composition*:

$$\frac{M : A / B \quad N : B / C}{M; N : A / C} \text{ cmp} \triangleq \frac{\frac{M : A / B \quad \overline{N v : B}}{M (N v) : A} /E}{\lambda v. M (N v) : A / C} /I^v$$

Now we are equipped to try our derivation again.

$$\begin{array}{c}
 \frac{\text{Lenin} : \text{dp} \quad \text{opposes} : \text{tv} \quad \text{and} : \text{conj}[?] \quad \text{Kerensky} : \text{dp} \quad \text{supports} : \text{tv} \quad \text{the} : \text{d} \quad \text{war} : \text{np}}{\text{Lenin} : \text{dp} \quad \text{opposes} : \text{tv} \quad \text{and} : \text{conj}[?] \quad \text{Kerensky} : \text{dp} \quad \text{supports} : \text{tv} \quad \text{the war} : \text{dp}} /E \\
 \hline
 \frac{\text{Lenin}^\dagger : \text{vp} / (\text{dp} \setminus \text{vp}) \quad \text{opposes} : (\text{dp} \setminus \text{vp}) / \text{dp} \quad \text{and} : \text{conj}[?] \quad \text{Kerensky} : \text{dp} \quad \text{supports} : \text{tv} \quad \text{the war} : \text{dp}}{\text{Lenin}^\dagger; \text{opposes} : \text{vp} / \text{dp} \quad \text{and} : \text{conj}[?] \quad \text{Kerensky} : \text{dp} \quad \text{supports} : \text{tv} \quad \text{the war} : \text{dp}} \text{raise} \\
 \hline
 \frac{\text{Lenin}^\dagger; \text{opposes} : \text{vp} / \text{dp} \quad \text{and} : \text{conj}[?] \quad \text{Kerensky}^\dagger : \text{vp} / (\text{dp} \setminus \text{vp}) \quad \text{supports} : \text{tv} \quad \text{the war} : \text{dp}}{\text{Lenin}^\dagger; \text{opposes} : \text{vp} / \text{dp} \quad \text{and} : (\text{vp} / \text{dp}) \setminus ((\text{vp} / \text{dp}) / (\text{vp} / \text{dp})) \quad \text{Kerensky}^\dagger; \text{supports} : \text{vp} / \text{dp} \quad \text{the war} : \text{dp}} \text{cmp} \\
 \hline
 \frac{\text{Lenin}^\dagger; \text{opposes} : \text{vp} / \text{dp} \quad \text{and} : (\text{vp} / \text{dp}) / (\text{vp} / \text{dp}) \quad \text{Kerensky}^\dagger; \text{supports} : \text{vp} / \text{dp} \quad \text{the war} : \text{dp}}{\text{Lenin}^\dagger; \text{opposes and Kerensky}^\dagger; \text{supports} : \text{vp} / \text{dp} \quad \text{the war} : \text{dp}} /E \\
 \hline
 \frac{\text{Lenin}^\dagger; \text{opposes and Kerensky}^\dagger; \text{supports} : \text{vp} / \text{dp} \quad \text{the war} : \text{dp}}{\text{Lenin}^\dagger; \text{opposes and Kerensky}^\dagger; \text{supports the war} : \text{vp}} /E
 \end{array}$$

**Exercise 2** Try to derive the each of following rules, or conjecture that it is impossible.

$$\frac{A}{X / (X / A)} \qquad \frac{A}{(X / A) \setminus X} \qquad \frac{A}{(A \setminus X) \setminus X}$$

### 2.2.3 Using Persistent Propositions

Consider adding a new atomic syntactic type  $\text{np}_{\text{mass}}$  for *mass nouns*. Among other things, mass nouns differ from ordinary nouns in that their determiner is silent (they do not need to use “the”): that is, this determiner appears in the parse tree, but it does not appear in the surface syntax.

This seems to poses a problem for parsing: we will not have the determiner in our state initially, since we don’t know in advance where we will need it. The solution is to regard this determiner as *persistent*, and simply add  $\emptyset : \text{dp} / \text{np}_{\text{mass}}$  to all our states. Then, whenever we need a mass noun determiner, we can freely add it in the appropriate spot.

Consider the sentence, “We demand peace and bread and land!”, where all three demands are mass nouns. To parse this, we want to derive a rule of the following shape:

$$\frac{\emptyset : \text{dp} / \text{np}_{\text{mass}} \quad \text{we} : \text{dp} \quad \text{demand} : \text{tv} \quad \text{peace} : \text{np}_{\text{mass}} \quad \text{and} : \text{conj}[\text{np}_{\text{mass}}] \quad \text{bread} : \text{np}_{\text{mass}} \quad \text{and} : \text{conj}[\text{np}_{\text{mass}}] \quad \text{land} : \text{np}_{\text{mass}}}{\emptyset : \text{dp} / \text{np}_{\text{mass}} \quad ??? : \text{vp}}$$

Stop and try to derive this before turning the page.

**Solution:**

$$\begin{array}{c}
 \frac{\emptyset : dp / np_{mass} \quad we : dp \quad demand : tv \quad peace : np_{mass} \quad and : conj[np_{mass}] \quad bread : np_{mass} \quad and : conj[np_{mass}] \quad land : np_{mass}}{\emptyset : dp / np_{mass} \quad we : dp \quad demand : tv \quad peace : np_{mass} \quad and : conj[np_{mass}] \quad bread and : np_{mass} / np_{mass} \quad land : np_{mass}} \backslash E \\
 \frac{}{\emptyset : dp / np_{mass} \quad we : dp \quad demand : tv \quad peace : np_{mass} \quad and : conj[np_{mass}] \quad bread and land : np_{mass}} / E \\
 \frac{}{\emptyset : dp / np_{mass} \quad we : dp \quad demand : tv \quad peace and : np_{mass} / np_{mass} \quad bread and land : np_{mass}} \backslash E \\
 \frac{}{\emptyset : dp / np_{mass} \quad we : dp \quad demand : tv \quad peace and bread and land : np_{mass}} / E \\
 \frac{\emptyset : dp / np_{mass} \quad we : dp \quad demand : tv \quad \emptyset : dp / np_{mass} \quad peace and bread and land : np_{mass}}{\emptyset : dp / np_{mass} \quad we : dp \quad demand : tv \quad \emptyset peace and bread and land : dp} / E \\
 \frac{}{\emptyset : dp / np_{mass} \quad we : dp \quad demand \emptyset peace and bread and land : iv} \backslash E \\
 \frac{}{\emptyset : dp / np_{mass} \quad we demand \emptyset peace and bread and land : vp} \backslash E
 \end{array}$$

## References

- [Bal02] Jason Baldridge. *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. PhD thesis, University of Edinburgh, 2002.
- [Lam58] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.
- [MMF04] K. Marx, E. Mandel, and B. Fowkes. *Capital: A Critique of Political Economy*. Number Volume 1 in Penguin classics. Penguin Books Limited, 2004.
- [Mor12] G.V. Morrill. *Type Logical Grammar: Categorial Logic of Signs*. Springer Netherlands, 2012.
- [MVF11] Glyn Morrill, Oriol Valentín, and Mario Fadda. The displacement calculus. *Journal of Logic, Language and Information*, 20(1):1–48, Jan 2011.
- [Pfe04] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In W.-N. Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, page 196, Taipei, Taiwan, November 2004. Springer-Verlag LNCS 3302. Abstract of invited talk.
- [PP] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. Submitted. Revised and extended version of abstract from TLCA'99.
- [PP98] Jeff Polakow and Frank Pfenning. Ordered linear logic programming. Technical Report CMU-CS-98-183, Department of Computer Science, Carnegie Mellon University, December 1998.
- [PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS 2009)*, pages 101–110, Los Angeles, California, August 2009. IEEE Computer Society Press.
- [Ste96] Mark Steedman. *Surface structure and interpretation*. Linguistic inquiry monographs, 30. MIT Press, 1996.

# Lecture Notes on Ordered Logic

15-317: Constructive Logic  
Frank Pfenning

Lecture 21  
November 16, 2017

## 1 Introduction

In this lecture we first review ordered inference with two new examples: a finite state transducer (which indicates how to represent finite state transducers in general), and Turing machines.

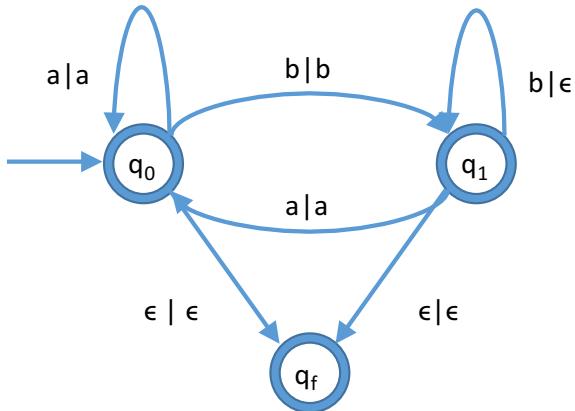
After that, we follow the playbook of Part II of this course and develop a sequent calculus so that all inference will proceed in one direction, namely bottom-up. We briefly consider some cases in the proof of cut admissibility for the system and carry out some sample proofs. For more on ordered logic, we recommend a course on [Substructural Logics](#) from Fall 2016 at Carnegie Mellon University.

Finally, we consider *linear logic* [Gir87] as an intermediate point between ordered and intuitionistic logic via a simple change to ordered logic.

## 2 Example: Finite State Transducers

A *subsequential finite-state transducer* [Sch77] (FST) consists of a finite number of states, an input alphabet and an output alphabet, and a transition function  $\delta$  that takes a state and an input symbol to a new state and an output string. We also distinguish an initial state and a final state, from which no further transitions are possible. Finite state transducers have a number of important closure properties and are closely related to deterministic finite automata (DFAs). They are often depicted with transition diagrams. As an example we show a FST which transforms an input string consisting

of  $a$  and  $b$  symbols by compressing all runs of  $b$  into a single  $b$ . Each transition is labeled as  $x \mid w$  where  $x$  is either an input symbol or  $\epsilon$  (when the input string is empty) and  $w$  is a word over the output alphabet.



In order to represent this in the Lambek calculus so that ordered inference corresponds to computation, we introduce propositions  $a$  and  $b$  to represent the symbols, here shared between the input and output alphabets. We also have a proposition  $\$$  representing an endmarker and *reverse* the word<sup>1</sup>. For example, the string  $bbaba$  will be represented as the ordered antecedents  $\$ a b a b b$ . Furthermore, we have a new proposition for every state in the FST, here  $q_0$ ,  $q_1$ , and  $q_f$ . Initially, our antecedents will be populated by the representation of the input string followed by the initial state. In this example, we start with

$\$ a b a b b q_0$

We now present inference rules so that each ordered inference corresponds to a transition of the finite state transducer. In the premise we have the input (represented as a proposition) followed by the state; in the conclusion we have the new state followed by the output. The empty input string is represented by  $\$$ , which we need to write when we transition into the final

---

<sup>1</sup>for reasons that may nor may not become clear in a future lecture

state.

$$\frac{a \ q_0}{q_0 \ a} \quad \frac{b \ q_0}{q_1 \ b} \quad \frac{\$ \ q_0}{q_f \ \$}$$

$$\frac{a \ q_1}{q_0 \ a} \quad \frac{b \ q_1}{q_1} \quad \frac{\$ \ q_1}{q_f \ \$}$$

Since it is convenient, we add one more inference rule

$$\frac{}{q_f}$$

.

so that the overall computation with input word  $w$ , and initial state  $q_0$  to output  $v$  in final state  $q_f$  is modeled by inference

$$\frac{\begin{array}{c} \$ \ w^R \ q_0 \\ \vdots \\ q_f \ \$ \ v^R \end{array}}{\$ \ v^R}$$

where  $s^R$  represents the reversal of a string  $s$ . We could also fold the last step into the rules producing  $q_f$ , replacing  $q_f$  by the empty context.

You can see why we used an endmarker  $\$$ : unlike the usual assumption for finite-state transducers, ordered inference cannot depend on whether it takes place at the end of the context. This is because any ordered inference, by its very definition, applies to any consecutive part of the state. In the sequent calculus this is explicit in all the left rules that have arbitrary  $\Omega_L$  and  $\Omega_R$  surrounding the principal proposition of the inference. Trying to restrict this would lead to a breakdown in the sequent calculus.

We can use this construction to represent any subsequential finite-state transducer, with one inference rule for every transition. We will not develop the formal details, which are somewhat tedious but straightforward.

We can compose transducers the way we could compose functions. If transducer  $T_1$  transforms input  $w_0$  into  $w_1$  and  $T_2$  transforms  $w_1$  to  $w_2$ , then  $T_1 ; T_2$  transforms  $w_0$  to  $w_2$ . There is a construction on the automata-theoretic descriptions of transducers to show that  $T_1 ; T_2$  is indeed another finite-state subsequential transducer if  $T_1$  and  $T_2$  are.

Here, in the setting of ordered inference, we can easily represent the composition of transducers  $T_1 ; \dots ; T_n$  just by renaming the sets of states apart and then creating the initial state as

$$\$ \ w^R \ q_0^1 \dots q_0^n$$

where  $q_0^i$  is the initial state of FST  $T_i$ . As  $T_1$  starts to produce output, the configuration will have the form

$$\$ w_0^R q_k^1 w_1^R q_0^2 \dots q_0^n$$

At this point,  $T_2$  (represented by  $q_0^2$ ) can start to consume some of its input and produce its output, and so on. Effectively, we have a chain of transducers operating concurrently as long as enough input is available to each of them. Eventually, all of them will end up in their final state and we will end up with the final configuration  $\$ v^R$ .

### 3 Example: Turing Machines

In this section we generalize the construction from the previous section to represent Turing machines. We represent the contents of the unbounded tape of the Turing machine as a *finite* context

$$\$ \dots a_{-1} q a_0 a_1 \dots \$$$

with two endmarkers  $\$$ . The proposition  $q$  represents the current state of the machine, and we imagine it “looks to its right” so that the contents of the current cell would be  $a_0$ . The initial context for the initial state  $q_0$  is just

$$\$ q_0 a_0 \dots a_n \$$$

where  $a_0 \dots a_n$  is the input word written on the tape. Returning to the general case

$$\$ \dots a_{-1} q a_0 a_1 \dots \$$$

if the transition function for state  $q$  and symbol  $a_0$  specifies to write symbol  $a'_0$ , transition to state  $q'$ , and move to the *right*, then the next configuration would be

$$\$ \dots a_{-1} a'_0 q' a_1 \dots \$$$

This can easily be represented, in general, by the rule

$$\frac{q \ a}{a' \ q'} \text{ MR}$$

which we call MR for *move right*.

To see how to represent moving to the left, reconsider

$$\$ \dots a_{-1} q a_0 a_1 \dots \$$$

If we are supposed to write  $a'_0$ , transition to  $q'$ , and move to the left, the next state should be

$$\$ \dots q' a_{-1} a'_0 a_1 \dots \$$$

The corresponding rule would, using  $b$  for  $a_{-1}$ :

$$\frac{b \ q \ a}{q' \ b \ a'} \text{ML}_b$$

We would have such a rule for each  $b$  in the (fortunately finite) tape alphabet (which excludes the endmarker), or we could represent it schematically

$$\frac{x \ q \ a}{q' \ x \ a'} \text{ML}^*$$

except we would have a side condition that  $x \neq \$$ . We should also have rules that allow us to extend the tape by the designated blank symbol ‘ $\_$ ’ (which is part of the usual definition of Turing machines).

$$\frac{\$ \ q \ a}{\$ \ q' \ \_ \ a'} \text{ML}_{\$} \quad \frac{q \ \$}{q \ \_ \ \$} \text{ER}_q$$

Finally, if we are in a final state  $q_f$  from which no further transitions are possible, we can simply eliminate it from the configuration.

$$\frac{q_f}{\_} F$$

A somewhat more symmetric and elegant solution allows the tape head in state  $q$  (represented by the proposition  $q$ ) to be looking either right or left, represented by  $q \triangleright$  and  $\triangleleft q$ . When we look right and have to move left or vice versa, we just change the direction in which we are looking to implement the move. Then we get the following elegant set of rules, two for each possible transition, two extra ones for extending the tape, and two (if we like) for erasing the final state.

$$\begin{array}{ll} \frac{q \triangleright \ a}{a' \ q' \triangleright} \text{LRMR} & \frac{q \triangleright \ a}{\triangleleft q' \ a'} \text{LRML} \\ \frac{a \triangleleft \ q}{a' \ q' \triangleright} \text{LLMR} & \frac{a \triangleleft \ q}{\triangleleft q' \ a'} \text{LLML} \\ \frac{\triangleright \ \$}{\triangleright \ \_ \ \$} \text{ER} & \frac{\$ \triangleleft}{\$ \ \_ \ \triangleleft} \text{EL} \\ \frac{\triangleleft \ q_f}{\_} \text{FL} & \frac{q_f \triangleright}{\_} \text{FR} \end{array}$$

The initial configuration represented by the context

$$\$ q_0 \triangleright a_1 \dots a_n \$$$

and the final configuration as

$$\$ b_1 \dots b_k \$$$

and we go from the first to the last by a process of ordered inference.

Of course, a Turing machine may not halt, in which case inference would proceed indefinitely, never arriving at a quiescent state in which no inference is possible.

Our modeling of the Turing machine is here faithful in the sense that each step of the Turing machine corresponds to one inference. There is a small caveat in that we have to extent the tape with an explicit inference, while Turing machines are usually preloaded with a two-way infinite tape with blank symbols on them. But except for those little stutter-steps, the correspondence is exact.

Composition of Turing machines in this representation is unfortunately not as simple as for FSTs since the output is not produced piecemeal, going in one direction, but will be on the tape when the final state is reached. We would have to return the tape head (presumably in the final state) to the left end of the tape and then transition to the starting state of the second machine.

Both for finite-state transducers and Turing machines, nondeterminism is easy to add: we just add multiple rules if there are multiple possible transitions from a state. This works, because the inference process is naturally nondeterministic: any applicable rule can be applied.

We will return to automata and Turing machines in a future lecture when we will look at the problem again from a different perspective.

## 4 Ordered Hypothetical Judgments

The notion of grammatical inference represents parsing as the process of constructing a proof. For example, if we have a phrase (= sequence of words)  $w_1 \dots w_n$  we find their syntactical types  $x_1 \dots x_n$  (guessing if necessary if they are ambiguous) and then set up the problem

$$(w_1 : x_1) \cdots (w_n : x_n)$$

$$\begin{array}{c} \vdots \\ ? : s \end{array}$$

where “?” will represent the parse tree as a properly parenthesized expression (assuming, of course, we can find a proof).

So far, we can only represent the inference itself, but not the *goal* of parsing a whole sentence. In order to express that we introduce *hypothetical judgments* as a new primitive concept. The situation above is represented as

$$(w_1 : x_1) \cdots (w_n : x_n) \implies ? : s$$

or, more generally, as

$$(p_1 : x_1) \cdots (p_n : x_n) \implies r : z$$

The turnstile symbol “ $\implies$ ” here separates the *succedent*  $r : z$  from the *antecedents*  $p_i : x_i$ . We sometimes call the left-hand side the *context* or the *hypotheses* and the right-hand side the *conclusion*. Calling the succedent a conclusion is accurate in the sense that it is the conclusion of a hypothetical deduction, but it can also be confusing since we also used “conclusions” to describe what is below the line in a rule of inference. We hope it will always be clear from the situation which of these we mean.

Since we are studying ordered inference right now, the antecedents that form the context are intrinsically ordered. When we want to refer to a sequence of such antecedents we write  $\Omega$  where “Omega” is intended to suggest “Order”. When we capture other forms of inference like linear inference we will revisit this assumption.

## 5 Inference with Sequents: Looking Left

Now that we have identified hypothetical judgments, written as sequents  $\Omega \implies r : z$ , we should examine what this means for our logical rules of inference. Fortunately, we have had only two connectives, *over* and *under*, first shown here without the proof terms (that is, without the parse trees):

$$\frac{B / A \quad A}{B} /E \quad \frac{A \quad A \setminus B}{B} \setminus E$$

Now that the propositions we know appear as antecedents, the direction of the rules appears to be reversed when considered on sequents. Let us remember, by analogy, the elimination rule of natural deduction and the left rules for the sequent calculus for ordinary implication (without consideration of order)

$$\frac{A \supset B \quad A}{B} \supset E \quad \frac{\Gamma, A \supset B \implies A \quad \Gamma, A \supset B, B \implies C}{\Gamma, A \supset B \implies C} \supset L$$

In the ordered case, say  $A \setminus B$  the antecedents will have form

$$\Omega_1 (A \setminus B) \Omega_2$$

because we are allowed to apply the  $\setminus L$  rule to any antecedent. In the intuitionistic case, we just say  $\Gamma, A \supset B$  because we could silently reorder the antecedents.

The second consideration is that the proof of  $A$  must be from a block of antecedents that are immediately to the left of  $A \setminus B$ . This is because the premises of the  $\setminus E$  must be consecutive among the current hypothesis.

With these considerations, we arrive at the following rule

$$\frac{\Omega \Rightarrow A \quad \Omega_L B \Omega_R \Rightarrow C}{\Omega_L \Omega (A \setminus B) \Omega_R \Rightarrow C} \setminus L$$

We have written  $\Omega_L$  and  $\Omega_R$  to indicate the rest of the context, which remains unaffected by the inference. The left rule for “over” ( $B / A$ ) is symmetric:

$$\frac{\Omega_L B \Omega_R \Rightarrow C \quad \Omega \Rightarrow A}{\Omega_L (B / A) \Omega \Omega_R \Rightarrow C} / L$$

Our inferences, now taking place on the antecedent, take us upward in the tree. This means we need at least one more rule to complete the proof and signal the success of a hypothetical proof. Both forms with and without the proof terms should be self-explanatory. We use  $\text{id}$  (for *identity*) to label this inference.

$$\overline{A \Rightarrow A} \text{id}_A$$

Unlike the usual intuitionistic sequent calculus,  $A$  must be the only antecedent. This is akin to saying that we do not allow *weakening*, neither implicitly nor explicitly.

## 6 Inference with Sequents: Looking Right

Thinking about the parsing problem  $A \setminus (B / C)$  should be somehow equivalent to  $(A \setminus B) / C$  since both yield a  $B$  when given an  $A$  to the left and  $C$  to the right. Setting this equivalence up as two hypothetical judgments

$$A \setminus (B / C) \Rightarrow (A \setminus B) / C$$

and

$$(A \setminus B) / C \implies A \setminus (B / C)$$

that we are trying to prove however fails. No inference is possible. We are lacking the ability to express when we can deduce a *succedent* with a logical connective. Lambek [Lam58] states that we should be able to deduce

$$\frac{C}{B / A} \quad \text{if} \quad \frac{C \quad A}{B}$$

So  $B / A$  should follow from  $C$  if we get  $B$  if we put  $A$  to the right of  $C$ . With pure inference, as practiced in the last lecture, we had no way to turn this “if” into form of inference rule. However, armed with hypothetical judgments it is not difficult to express precisely this:

$$\frac{C \quad A \implies B}{C \implies B / A}$$

Instead of a single proposition  $z$  we allow a context, so we write this

$$\frac{\Omega \quad A \implies B}{\Omega \implies B / A} /R$$

This is an example of a *right rule*, because it analyzes the structure of a proposition in the succedent and we pronounce it as *over right*. The  $\setminus R$  (*under right*) rule can be derived analogously.

$$\frac{A \quad \Omega \implies B}{\Omega \implies A \setminus B} \setminus R$$

In the next section we will look at the question how we know that these rules are correct. For example, we might have accidentally swapped these two rules, in which case our logic would somehow be flawed.

Let's come back to the motivating example and try to construct a proof of

$$A \setminus (B / C) \implies (A \setminus B) / C$$

Remember, all the rules work bottom-up, either on some antecedent (a left rule) or on the succedent (a right rule). No left rule applies here (there is no  $x$  to the left of  $x \setminus (\dots)$ ) but fortunately the  $/R$  rule does.

$$\frac{(A \setminus (B / C)) \quad C \implies A \setminus B}{A \setminus (B / C) \implies (A \setminus B) / C} /R$$

Again, no left rule applies (the parentheses are in the wrong place) but a right rule does.

$$\frac{\begin{array}{c} A \quad A \setminus (B / C) \quad C \Rightarrow B \\ \hline A \setminus (B / C) \quad C \Rightarrow A \setminus B \end{array}}{A \setminus (B / C) \Rightarrow (A \setminus B) / C} \setminus R$$

Finally, now a left rule applies.

$$\frac{\begin{array}{c} \overline{A \Rightarrow A} \quad \text{id} \quad (B / C) \quad C \Rightarrow B \\ \hline A \quad (A \setminus (B / C)) \quad C \Rightarrow B \end{array}}{\frac{\begin{array}{c} \overline{(A \setminus (B / C))} \quad C \Rightarrow A \setminus B \\ \hline (A \setminus (B / C)) \Rightarrow (A \setminus B) / C \end{array}}{(A \setminus (B / C)) \Rightarrow (A \setminus B) / C} / R} \setminus L$$

One more left rule, and then we can apply identity.

$$\frac{\begin{array}{c} \overline{A \Rightarrow A} \quad \text{id} \quad \frac{\overline{B \Rightarrow B} \quad \text{id}_B \quad \overline{C \Rightarrow C} \quad \text{id}_C}{(B / C) \quad C \Rightarrow B} / L \\ \hline A \quad (A \setminus (B / C)) \quad C \Rightarrow B \end{array}}{\frac{\begin{array}{c} \overline{(A \setminus (B / C))} \quad C \Rightarrow A \setminus B \\ \hline (A \setminus (B / C)) \Rightarrow (A \setminus B) / C \end{array}}{(A \setminus (B / C)) \Rightarrow (A \setminus B) / C} / R} \setminus L$$

The proof in the other direction is similar and left as exercise.

## 7 Alternative Conjunction

As already mentioned in the last lecture, some words have more than one syntactic type. For example, *and* has type  $s \setminus s / s$  (omitting parentheses now since the two forms are equivalent by the reasoning the previous section) and also type  $n \setminus n^* / n$ , constructing a plural noun from two singular ones. We can combine this into a single type  $x \& y$ , pronounced *x with y*:

$$\textit{and} : (s \setminus s / s) \& (n \setminus n^* / n)$$

Then, in a deduction, we are confronted with a choice between the two for every occurrence of *and*. For example, in typing *Alice and Bob work and Eve*

likes Alice, we choose  $n \setminus n^* / n$  for the first occurrence of *and*, and  $s \setminus s / s$  for the second.

Lambek did not explicitly define this connective, but it would be defined by the rules

$$\frac{A \& B}{A} \&E_1 \quad \frac{A \& B}{B} \&E_2$$

As before, these rules turn into left rules in the sequent calculus, shown here only without the proof terms.

$$\frac{\Omega_L A \Omega_R \Rightarrow C}{\Omega_L (A \& B) \Omega_R \Rightarrow C} \&L_1 \quad \frac{\Omega_L B \Omega_R \Rightarrow C}{\Omega_L (A \& B) \Omega_R \Rightarrow C} \&L_2$$

To derive the right rule we must ask ourselves under which circumstances we could use a proposition both as an  $x$  and as a  $y$ . That's true, if we can show both, from the same antecedents.

$$\frac{\Omega \Rightarrow A \quad \Omega \Rightarrow B}{\Omega \Rightarrow A \& B} \&R$$

## 8 Concatenation

In a sequent, there are multiple antecedents (in order!) but only one succedent. If we need to consider multiple propositions in the succedent we need to define a new connective that expresses adjacency as a new proposition. We write  $x \bullet y$  (read:  $x$  fuse  $y$ ). In the Lambek calculus, we would simply write

$$\frac{A \bullet B}{A \ B} \bullet E$$

As a left rule, this is simple turned upside down and becomes

$$\frac{\Omega_L x \ y \ \Omega_R \Rightarrow z}{\Omega_L x \bullet y \ \Omega_R \Rightarrow z} \bullet L$$

As a right rule for  $x \bullet y$ , we have to divide the context into two segments, one proving  $x$  and the other proving  $y$ .

$$\frac{\Omega_1 \Rightarrow x \quad \Omega_2 \Rightarrow y}{\Omega_1 \ \Omega_2 \Rightarrow x \bullet y} \bullet R$$

Note that there is some nondeterminism in this rule if we decide to use it to prove a sequent, because we have to decide *where* to split the context

$\Omega = (\Omega_1 \ \Omega_2)$ . For a context with  $n$  propositions there are  $n + 1$  possibilities. For example, if we want to express that a phrase represented by  $\Omega$  is parsed into *two sentences* we can prove the hypothetical judgment

$$\Omega \implies s \bullet s$$

We can then prove

$$\begin{array}{ccccccc} Alice & works & Bob & sleeps & & ? \\ : & : & : & : & & : \\ n & n \setminus s & n & n \setminus s & \implies & s \bullet s \end{array}$$

but we have to split the phrase exactly between *works* and *Bob* so that both premises can be proved. Assuming a notation of  $p \cdot q : x \bullet y$  if  $p : x$  and  $q : y$ , the proof term for  $s \bullet s$  in this example would be  $(Alice \text{ works}) \cdot (Bob \text{ sleeps})$ .

## 9 Emptiness<sup>2</sup>

In this section we consider **1**, the unit of concatenation, which corresponds to the empty context. The left and right rules are nullary versions of the binary concatenation. In particular, there must be no antecedents in the right rule for **1**.

$$\frac{}{\implies \mathbf{1}} \mathbf{1R} \quad \frac{\Omega_L \ \Omega_R \implies z}{\Omega_L \ \mathbf{1} \ \Omega_R \implies z} \mathbf{1L}$$

## 10 Admissibility of Cut

We return from the examples to metatheoretic considerations. Our goal in this section and the next is to show that the cut rule is admissible in the ordered sequent calculus. Together with identity elimination this gives us a global version of harmony for our logic and a good argument for thinking of the right and left rules in the sequent calculus as defining the meaning of the connectives.

A key step on the way will be the *admissibility of cut* in the cut-free sequent calculus. We say that a rule of inference is *admissible* if there is a proof of the conclusion whenever there are proofs of all the premises. This is a somewhat weaker requirement than saying that a rule is *derivable*, which means we have a closed-form hypothetical proof of the conclusion given all

---

<sup>2</sup>not covered in lecture

the premises. Derivable rules remain derivable even if we extend our logic by new propositions and inference rules (once a proof, always a proof), but admissible rules may no longer remain admissible and have to be reconsidered.

Since the cut-free sequent calculus will play an important role in this course, we write  $\Omega \Rightarrow x$  for a sequent in the cut-free sequent calculus. We write admissible rules using dashed lines and parenthesized justifications, as in

$$\frac{\Omega \Rightarrow A \quad \Omega_L A \Omega_r \Rightarrow C}{\Omega_L \Omega \Omega_R \Rightarrow C} \text{ (cut}_A\text{)}$$

Of course, we have not yet proved that cut is indeed admissible here! It turns out that the proof remains essentially the same as we have seen for intuitionistic logic.

### Theorem 1 (Admissibility of Cut)

If  $\Omega \Rightarrow A$  and  $\Omega_L A \Omega_R \Rightarrow C$  then  $\Omega_L \Omega \Omega_R \Rightarrow C$ .

**Proof:** We assume we are given  $\Omega \xrightarrow{\mathcal{D}} A$  and  $\Omega_L A \Omega_R \xrightarrow{\mathcal{E}} C$  and we construct  $\Omega_L \Omega \Omega_R \xrightarrow{\mathcal{F}} C$ .

The proof proceeds by a so-called *nested induction*, first on  $A$  and then the proofs  $\mathcal{D}$  and  $\mathcal{E}$ . This means we can appeal to the induction hypothesis when

1. either the cut formula  $A$  becomes smaller,
2. or  $A$  remains the same, and
  - (a)  $\mathcal{D}$  becomes smaller and  $\mathcal{E}$  stays the same,
  - (b) or  $\mathcal{D}$  stays the same and  $\mathcal{E}$  becomes smaller.

This is also called *lexicographic induction* since it is an induction over a lexicographic order, first considering  $x$  and then  $\mathcal{D}$  and  $\mathcal{E}$ .

The idea for this kind of induction can be synthesized from the proof if we observe what constructions take place in each case. We will see that the ideas of the cut reductions in the last lecture will be embodied in the proof cases. We distinguish three kinds of cases based on  $\mathcal{D}$  and  $\mathcal{E}$ .

**Identity cases.** When one premise or the other is an instance of the identity rule we can eliminate the cut outright. This should be expected since identity (“if we can use  $x$  we may prove  $x$ ”) and cut (“if we can prove  $x$  we may use  $x$ ”) are direct inverses of each other.

**Principal cases.** When the cut formula  $x$  is introduced by the last inference in both premises we can reduce the cut to (potentially several) cuts on strict subformulas of  $A$ . We have demonstrated this by cut reductions in the last lecture.

**Commutative cases.** When the cut formula is a side formula of the last inference in either premise, we can appeal to the induction hypothesis on this premise and then re-apply the last inference. These constitute valid appeals to the induction hypothesis because the cut formula and one of the deductions in the premises remain the same while the other becomes smaller.

We now go through representative samples of these cases. First, the two identity cases.<sup>3</sup>

**Case:**  $\text{id} \# \mathcal{E}$

$$\mathcal{D} = \frac{}{A \Rightarrow A} \text{id}_A \quad \text{and} \quad \Omega_L A \Omega_R \Rightarrow C \quad \text{arbitrary}$$

We have to construct a proof of  $\Omega_L \Omega \Omega_R \Rightarrow C$ , but  $\Omega = A$ , so we can let  $\mathcal{F} = \mathcal{E}$ .

**Case:**  $\mathcal{D} \# \text{id}$

$$\Omega \Rightarrow A \quad \text{arbitrary, and} \quad \mathcal{E} = \frac{\mathcal{D}}{A \Rightarrow A} \text{id}_A$$

We have to construct a proof of  $\Omega_L \Omega \Omega_R \Rightarrow C$ , but  $\Omega_L = \Omega_R = (\cdot)$  and  $C = A$ , so we can let  $\mathcal{F} = \mathcal{D}$ .

Next we look at a principal case, where the cut proposition  $x$  (here  $x_1 / x_2$ ) was introduced in the last inference in both premises, in which case we say  $x$  is the *principal formula* of the inference.

**Case:**  $/R \# /L$

$$\mathcal{D} = \frac{\Omega A_2 \Rightarrow A_1}{\Omega \Rightarrow A_1 / A_2} /R \quad \text{and} \quad \mathcal{E} = \frac{\Omega'_R \Rightarrow A_2 \quad \Omega_L A_1 \Omega''_R \Rightarrow z}{\Omega_L (A_1 / A_2) \Omega'_R \Omega''_R \Rightarrow z} /L$$

---

<sup>3</sup>in lecture, we only showed a couple of principal cases

Using the intuition gained from cut reduction, we can apply the induction hypothesis on  $A_2$ ,  $\mathcal{E}_2$ , and  $\mathcal{D}_1$  and we obtain

$$\Omega \Omega'_R \xrightarrow{\mathcal{D}'_1} A_1 \quad \text{by i.h. on } A_2, \mathcal{E}_2, \mathcal{D}_1$$

We can once again apply the induction hypothesis, this time on  $A_1$ ,  $\mathcal{D}'_1$ , and  $\mathcal{E}_1$ :

$$\Omega_L \Omega \Omega'_R \Omega''_R \xrightarrow{\mathcal{E}'_1} z \quad \text{by i.h. on } A_1, \mathcal{D}'_1, \mathcal{E}_1$$

Note that  $\mathcal{D}'_1$  is the result of the previous appeal to the induction hypothesis and therefore not known to be smaller than  $\mathcal{D}_1$ , but the appeal to the induction hypothesis is justified since  $A_1$  is a subformula of  $A_1 / A_2$ .

Now we can let  $\mathcal{F} = \mathcal{E}'_1$  since  $\Omega_R = \Omega'_R \Omega''_R$  in this case, so we already have the right endsequent.

A more concise way to write down the same argument is in the form of a tree, where rules that are admissible (by induction hypothesis!) are justified in this manner.

Given

$$\frac{\begin{array}{c} \mathcal{D}_1 \\ \Omega A_2 \xrightarrow{} A_1 \\ \hline \Omega \xrightarrow{} A_1 / A_2 \end{array} /R \quad \frac{\begin{array}{c} \mathcal{E}_2 \\ \Omega'_R \xrightarrow{} A_2 \\ \Omega_L A_1 \Omega''_R \xrightarrow{} z \end{array} /L}{\Omega_L A_1 / A_2 \Omega'_R \Omega''_R \xrightarrow{} z}}{\Omega_L \Omega \Omega'_R \Omega''_R \xrightarrow{} z} \text{ (cut?)}$$

construct

$$\frac{\begin{array}{c} \mathcal{E}_2 \\ \Omega'_R \xrightarrow{} A_2 \\ \Omega A_2 \xrightarrow{} A_1 \\ \hline \Omega \Omega'_R \xrightarrow{} A_1 \end{array} \text{ (i.h. on } A_2\text{)} \quad \frac{\begin{array}{c} \mathcal{D}_1 \\ \Omega_L A_1 \Omega''_R \xrightarrow{} z \end{array} \text{ (i.h. on } A_1\text{)}}{\Omega_L \Omega \Omega'_R \Omega''_R \xrightarrow{} z}}$$

This is of course the local reduction, revisited as part of an inductive proof.

Finally we look at a *commutative case*, where the last inference rule applied in the first or second premise of the cut must have been different from the cut formula. We call this a *side formula*. We organize the cases around which rule was applied to which premise. Fortunately, they all go the same way: we “push” up the cut past the inference that was applied to the side formula. We show only one example.

**Case:**  $\mathcal{D} \# \bullet R$

$$\Omega \xrightarrow{\mathcal{D}} A \quad \text{arbitrary} \quad \text{and} \quad \mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \\ \Omega'_L \xrightarrow{\mathcal{D}} C_1 \quad \Omega''_L A \Omega_R \xrightarrow{\mathcal{E}_2} C \end{array}}{\Omega'_L \Omega''_L A \Omega_R \xrightarrow{\mathcal{E}} C_1 \bullet C_2} \bullet R$$

In this case we have the situation

$$\frac{\Omega \xrightarrow{\mathcal{D}} A \quad \frac{\begin{array}{c} \mathcal{E}_1 \\ \Omega'_L \xrightarrow{\mathcal{D}} C_1 \quad \Omega''_L A \Omega_R \xrightarrow{\mathcal{E}_2} C \end{array}}{\Omega'_L \Omega''_L A \Omega_R \xrightarrow{\mathcal{E}} C_1 \bullet C_2} \bullet R}{\Omega'_L \Omega''_L \Omega \Omega_R \xrightarrow{\mathcal{E}} C_1 \bullet C_2} \text{ (cut?)}$$

and construct

$$\frac{\begin{array}{c} \mathcal{D} \\ \Omega \xrightarrow{\mathcal{D}} A \quad \Omega''_L A \Omega_R \xrightarrow{\mathcal{E}_2} C \end{array}}{\Omega'_L \xrightarrow{\mathcal{E}_1} C_1 \quad \frac{\Omega''_L \Omega \Omega_R \xrightarrow{\mathcal{E}} C_2}{\Omega'_L \Omega''_L \Omega \Omega_R \xrightarrow{\mathcal{E}} C_1 \bullet C_2}} \bullet R \quad \text{i.h. on } A, \mathcal{D}, \mathcal{E}_2$$

Effectively, we have commuted the cut upward, past the  $\bullet R$  inference.

□

Our proof was *constructive*: it presents an effective method for constructing a cut-free proof of the conclusion, given cut-free proofs of the premises. The algorithm that can be extracted from the proof is nondeterministic, since some of the commuting cases overlap when the principal formula is a side formula in both premises. For most logics (although usually classical logic) the result is unique up to further permuting conversions between inference rules, a characterization we will have occasion to discuss later.

## 11 Linear Logic

In ordered logic, we use *ordered antecedents* which we have previously already used to make the inversion phase of proof search deterministic. We can define

$$\Omega ::= \epsilon \mid A \mid \Omega_1 \cdot \Omega_2$$

where ‘·’ is an associative operator with unit  $\epsilon$ . In the lecture so far, we have omitted  $\cdot$  and just used juxtaposition, and replace  $\epsilon$  by the empty context.

Now we obtain *linear logic* by using antecedents  $\Delta$  of the form

$$\Delta ::= \epsilon \mid A \mid \Delta_1, \Delta_2$$

where ‘,’ is an associate and *commutative* operator with unit  $\epsilon$ . Otherwise, the rules stay exactly the same as those for ordered logic!

Now we notice some interesting phenomena. For example, in ordered logic we have two implications,  $A \setminus B$  and  $B / A$ . But if order does not matter, we cannot tell which side of the antecedents  $A$  will end up at so the two become logically equivalent. We write  $A \multimap B$  (pronounced *A lilli B*).

Conjunction is even more interesting. In ordered logic we have three forms of conjunction  $A \bullet B$  (*A fuse B*), and  $A \circ B$  (*A twist B*, see [Problem 2, Assignment 9](#)) and  $A \& B$  (*A with B*). In linear logic, fuse and twist collapse to  $A \otimes B$  (*A tensor B*) because their only difference is the order of the components. On the other hand  $A \& B$  remains the same and is available in linear as well as ordered logic. Finally, in intuitionistic logic there is only one conjunction,  $A \wedge B$ , when one considers provability. But, if one considers the structure of proofs as well, we actually have to: a negative one (corresponding to  $A \& B$ ) and a positive one ( $A \otimes B$ ).

We will discuss linear logic and its operational interpretation further in the upcoming lectures.

## References

- [Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [Lam58] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.
- [Sch77] Marcel Paul Schützenberger. Sur une variante des fonctions séquentielles. *Theoretical Computer Science*, 4(1):47–57, 1977.

# Lecture Notes on Subsingleton Logic

15-317: Constructive Logic  
Frank Pfenning

Lecture 23  
November 28, 2017

## 1 Introduction

In this lecture we will first define *subsingleton logic* which is the subset of ordered logic where each judgment has at most one antecedent. Then we provide an *operational interpretation* of subsingleton logic in which *cut reduction* drives computation. This is in contrast to what we have been doing so far, where *logical inference*, that is, *proof construction* models computation.

The correspondence from this lecture is summarized in the following table.

Logic	Programming
Propositions	Session types
Ordered proofs	Concurrent programs
Cut reduction	Communication

This is an instance of a very general connection between proofs and programs studied in type theory. We can vary the logic and the computational interpretation. The big upside of this form of correspondence is that it helps us design programming languages in concert with the logic for reasoning about its programs.

This analysis was pioneered by Curry [Cur34] who related proofs in axiomatic form with combinatory logic. Late, Howard [How69] made the discovery that the Church's simply typed  $\lambda$ -calculus was in bijective correspondence with intuitionistic natural deduction. The particular instance of this correspondence for subsingleton logic is a recent discovery by DeYoung and yours truly [DP16].

## 2 Subsingleton Logic

We can examine the rules for each of the connectives to see which of them are still meaningful if we restrict ourselves to at most one antecedent. For example,

$$\frac{\Omega \ A \vdash B}{\Omega \vdash B / A} /R$$

would have a premise with two antecedents if the conclusion has only one.

What remains from the connectives we have introduced so far is only  $x \& y$  and  $\mathbf{1}$ . But we haven't had a notion of *disjunction* yet, which is written as  $x \oplus y$ . It is a disjunction, which means that  $x \oplus y$  is true if either  $x$  or  $y$  is true. So we have two right rules:

$$\frac{\Omega \vdash A}{\Omega \vdash A \oplus B} \oplus R_1 \quad \frac{\Omega \vdash B}{\Omega \vdash A \oplus B} \oplus R_2$$

Knowing that  $x$  or  $y$  is true, but not which one, means that the left rule proceeds by cases.

$$\frac{\Omega_L \ A \ \Omega_R \vdash C \quad \Omega_L \ B \ \Omega_R \vdash C}{\Omega_L \ (A \oplus B) \ \Omega_R \vdash C} \oplus L$$

It is straightforward to check the identity expansion and cut reduction, as well as extend the proof of cut elimination accordingly. Disjunction, just like the alternative conjunction, make sense in subsingleton logic.

We summarize the rules of subsingleton logic, with two small notational changes: we write  $\omega$  for zero or one antecedent.

$$\begin{array}{c} \frac{}{A \vdash A} \text{id}_A \quad \frac{\omega \vdash A \quad A \vdash C}{\omega \vdash C} \text{cut}_A \\ \\ \frac{\omega \vdash A}{\omega \vdash A \oplus B} \oplus_1 \quad \frac{\omega \vdash B}{\omega \vdash A \oplus B} \oplus_2 \quad \frac{A \vdash C \quad B \vdash C}{A \oplus B \vdash C} \oplus L \\ \\ \frac{\omega \vdash A \quad \omega \vdash B}{\omega \vdash A \& B} \& R \quad \frac{A \vdash C}{A \& B \vdash C} \& L_1 \quad \frac{B \vdash C}{A \& B \vdash C} \& L_2 \\ \\ \frac{}{\cdot \vdash \mathbf{1}} \mathbf{1} R \quad \frac{\cdot \vdash C}{\mathbf{1} \vdash C} \mathbf{1} L \end{array}$$

### 3 Proofs as Programs

We write  $\omega \vdash P : A$  with two alternative interpretations:

1.  $P$  is a proof of  $A$  with antecedent  $\omega$ .
2.  $P$  is a process providing  $A$  and using  $\omega$ .

Two processes are composed by cut, so that if

$$\omega \vdash P : A \quad A \vdash Q : C$$

then  $P$  and  $Q$  can run next to each other and exchange messages. Which messages can be exchanged is dictated by the type (= proposition)  $A$ .

As an example, consider

$$\omega \vdash P : A \oplus B \quad A \oplus B \vdash Q : C$$

The proof  $P$  of  $A \oplus B$  will contain critical information, namely if  $A$  is true or if  $B$  is true. Since the proof  $Q$  of  $C$  must account for both possibilities, we see that  $P$  will eventually *send* some information (inl if  $A$  is true, or inr if  $B$  is true) and  $Q$  will *receive* it. In a synchronous communication model, the message exchange can only take place if both sides are ready, which correspond to a principal case of cut where  $A \oplus B$  is the principal formula in both inferences. Using this intuition to fill in proof terms for one of the cases we arrive at:

$$\frac{\omega \vdash P : A}{\omega \vdash (\text{R.inl} ; P) : A \oplus B} \oplus R_1 \quad \frac{A \vdash Q_1 : C \quad B \vdash Q_2 : C}{A \oplus B \vdash (\text{caseL (inl} \Rightarrow Q_1 \mid \text{inr} \Rightarrow Q_2)) : C} \oplus L$$

This reduces to the new cut at type  $A$

$$\omega \vdash P : A \quad A \vdash Q_1 : C$$

So we think of  $(\text{R.inl} ; P)$  as sending the label inl to the right and then continuing as  $P$ , while  $\text{caseL (inl} \Rightarrow Q_1 \mid \text{inr} \Rightarrow Q_2)$  receives either inl or inr from the left and continues as  $Q_1$  or  $Q_2$ , respectively. The types  $A$  that type the interface between two processes are called *session types* (see [HHN<sup>+</sup>14] for a survey). A strong logical foundation for session types in *linear logic* was discovered by Caires and your lecturer [CP10] and later extended by others [Wad12, CPT13, Ton15].

If  $\oplus R_2$  was used in the first proof, then the new cut would be at type  $B$ . In either case, the communication corresponds exactly to a principal case in cut reduction.

Looking at computation more globally, processes are configured into a linear chain

$$P_1 \mid P_2 \mid P_3 \mid \dots \mid P_n$$

Such a configuration is well-typed if we have

$$(\omega_0 \vdash P_1 : A_1) \quad (A_1 \vdash P_2 : A_2) \quad (A_2 \vdash P_3 : A_3) \quad \dots \quad (A_n \vdash P_n : A_n)$$

where for any two adjacent processes, the type  $A_i$  provided by  $P_i$  has to be the same as the one used by  $P_{i+1}$ .

We now go through the rules and connective of ordered logic and develop the operational interpretation of proofs.

**Cut as Composition.** Cut is straightforward, since it just corresponds to parallel composition.

$$\frac{\omega \vdash P : A \quad A \vdash Q : C}{\omega \vdash (P \mid Q) : C} \text{ cut}_A$$

Computationally, a process  $P \mid Q$  will spawn  $P$  and continue as  $Q$  which might written as

$$\frac{(P \mid Q)}{P \mid Q}$$

Clearly, this rule preserves the typing invariant for a configuration if  $(P \mid Q)$  is typed by the cut rule, since the type  $A$  provided by  $P$  is exactly the same as the one as used by  $Q$ , and the left and right interfaces  $\omega$  and  $C$ , respectively, are preserved.

**Identity as Forwarding.** Cut creates two processes from one, while identity removes one process from the configuration, acting like a “forwarding” between the processes to its sides.

$$\frac{}{A \vdash \leftrightarrow : A} \text{id}_A$$

The computation rule simply removes the process from the configuration.

$$\frac{}{\cdot} \leftrightarrow$$

Again, this preserves the typing invariant for configurations since the processes to the left and right of ( $\leftrightarrow$ ) have the same type  $A$  on the right and left sides, respectively.

Now we come to the logical connectives. We already foreshadowed the case for disjunction, but we first generalize it to be more amenable for programming without changing its logical meaning.

**Disjunction as Internal Choice.** We generalize disjunction to be an  $n$ -ary connective by written  $\oplus\{l_i : A_i\}_{i \in I}$  for some finite index set  $I$  and *labels*  $l_i$ . Now the binary disjunction is defined as  $A \oplus B = \oplus\{\text{inl} : A, \text{inr} : B\}$ . Disjunction is also called *internal choice* since the proof itself determines which of the alternatives is chosen.

The right rule will send the appropriate label while the left rule will receive it and branch on it.

$$\frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash (\text{R}.l_k ; P) : \oplus\{l_i : A_i\}_{i \in I}} \oplus R_k \quad \frac{A_i \vdash Q_i : C \quad (\text{for all } i \in I)}{\oplus\{l_i : A_i\}_{i \in I} \vdash \text{caseL}(l_i \Rightarrow Q_i)_{i \in I} : C} \oplus L$$

Again, the computation rule just mirrors the cut reduction and therefore is easily seen to preserve configuration typing.

$$\frac{(\text{R}.l_k ; P) \mid (\text{caseL}(l_i \Rightarrow Q_i)_{i \in I})}{P \mid Q_k} \oplus C$$

The interface type between the two adjacent processes transitions from  $\oplus\{l_i : A_i\}_{i \in I}$  to  $A_k$  for some  $k \in I$ .

**Alternative Conjunction as External Choice.** Again, we generalize from  $A \& B$  to  $\&\{l_i : A_i\}_{i \in I}$  and defined  $A \& B = \&\{\text{inl} : A, \text{inr} : B\}$ .  $A \& B$  is sometimes called *external choice* since its proof must account for both possibilities and the clients selects between them. Otherwise, it is the straightforward dual of  $\oplus$ , sending to the left and receiving from the right.

$$\frac{\omega \vdash P_i : A_i \quad (\text{for all } i \in I)}{\omega \vdash \text{caseR}(l_i \Rightarrow P_i)_{i \in I} : \&\{l_i : A_i\}_{i \in I}} \& R \quad \frac{A_k \vdash Q : C \quad (k \in I)}{\&\{l_i : A_i\}_{i \in I} \vdash (\text{L}.l_k ; Q) : C} \& L_k$$

Again, the computation rule just mirrors the cut reduction and therefore is easily seen to preserve configuration typing.

$$\frac{(\text{caseR}(l_i \Rightarrow P_i)_{i \in I}) \mid (\text{L}.l_k ; Q)}{P_k \mid Q} \& C$$

**Unit as Termination.** The unit  $\mathbf{1}$  just corresponds to termination. Since communication is synchronous, the paired process to the right just waits for the termination to occur.

$$\frac{}{\cdot \vdash \text{closeR} : \mathbf{1}} \mathbf{1}^R \quad \frac{\cdot \vdash Q : C}{\mathbf{1} \vdash \text{waitL} ; Q : C} \mathbf{1}^L$$

The computation rule lets the waiting process proceed while the closing one disappears.

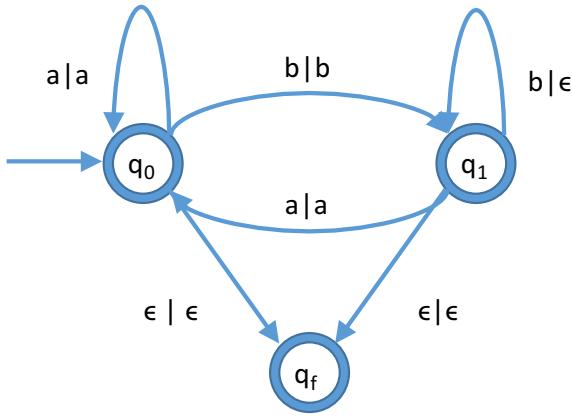
$$\frac{\text{closeR} \mid (\text{waitL} ; Q)}{Q}$$

This preserves types since there is no left interface to the configuration before and after the step, and the right interface  $C$  is preserved. Again, this can simply be read off from the cut reduction.

This completes the introduction of the computational interpretation of subsingleton logic. We will not discuss the computational metatheory, namely the *progress* and *preservation* theorems guarantee. Briefly, preservation here means that the configuration of process will remain well-typed during the computation, and progress that at least one process in a closed configuration can always take a step unless we are communicating and the right end of the configuration.

## 4 Example: Subsequential Finite State Transducers

We have already introduced FSTs in [Lecture 21](#), where we provided an implementation using the computation-as-ordered-inference paradigm. Here, we will use the computation-as-ordered-proof-reduction paradigm instead. We begin with the example of an FST that compresses runs of  $b$  into a single  $b$ .



We would like to represent the transducer as a process  $T$  that receives the input string, symbol by symbol, from the left and sends the output string, again symbol by symbol, to the right.

$$\text{string} \vdash T : \text{string}$$

The first problem is how to represent the type  $\text{string}$ . It is easy to represent symbols as *labels* and a choice between symbols  $a$ ,  $b$ , and the endmarker  $\$$  as an internal choice

$$\oplus\{a : A_a, b : A_b, \$ : A_{\$}\} \vdash T : \text{string}$$

Clearly,  $T$  can proceed with a  $\oplus L$  rule, which means it can branch on whether it receives an  $a$ ,  $b$ , or a  $\$$ . After receiving an  $a$ , for example, what should the type on the left be? We can receive further symbols, so it should be again  $\text{string}$ . This leads us to

$$\text{string} = \oplus\{a : \text{string}, b : \text{string}, \$ : A_{\$}\}$$

which is an example of a *recursive type* since  $\text{string}$  is defined in terms of itself. What should the remaining unspecified type  $A_{\$}$  be? Once we receive

the endmarker we can receive no further symbols (or anything else) from the left, we can only wait for the process to our left (that produced the string) to terminate. So  $A_{\$} = \mathbf{1}$  and we have

$$\text{string} = \oplus\{a : \text{string}, b : \text{string}, \$ : \mathbf{1}\}$$

It is convenient in this setting to think of the *string* as being *equal* to the type on the right. If every type definition is *contractive* [GH05] in the sense that it starts with a type constructor ( $\oplus, \&, \mathbf{1}$ ) then we do not need any explicit right or left rules since we can “silently” replace a type by its definition and apply the appropriate rule. This is the idea behind *equirecursive* treatment of recursive types. One should be worried that they could destroy all the good properties of the logic, but with some care this does not have to be the case.

Here is a simple program that produces the string *babb*:

$$\begin{aligned} \mathbf{1} &\vdash \lceil babb \rceil : \text{string} \\ \lceil babb \rceil &= R.b ; R.a ; R.b ; R.b ; R.\$ ; \leftrightarrow \end{aligned}$$

To deal with recursive types, the program will have to be similarly recursive. At the level of proofs, this can be analyzed as circular proofs [FS13], fixed points [Bae12], or corecursive proofs [TCP14]. We will just freely use recursion. Each state of the FST becomes a process definition that captures how the FST behaves with the corresponding input. Output is handled simply by sending the appropriate label to the right, and the new state is handled by invoking this state.

$$\begin{aligned} Q_0 &= \text{caseL } (a \Rightarrow R.a ; Q_0 \\ &\quad | b \Rightarrow R.b ; Q_1 \\ &\quad | \$ \Rightarrow R.\$ ; Q_f) \\ Q_1 &= \text{caseL } (a \Rightarrow R.a ; Q_0 \\ &\quad | b \Rightarrow Q_1 \\ &\quad | \$ \Rightarrow R.\$ ; Q_f) \\ Q_f &= \leftrightarrow \end{aligned}$$

The type of the final state  $Q_f$  is a bit different, since we know input and output have completed by the time this state is reached. We have

$$\begin{aligned} \text{string} &\vdash Q_0 : \text{string} \\ \text{string} &\vdash Q_1 : \text{string} \\ \mathbf{1} &\vdash Q_f : \mathbf{1} \end{aligned}$$

We also note an alternative definition for  $Q_f$

$$Q_f = \text{waitL} ; \text{closeR}$$

These two definitions are equivalent in the sense that  $(\text{waitL} ; \text{closeR})$  is the identity expansion of  $\leftrightarrow : \mathbf{1}$ . We will not go into detail, but this means that those two processes are *observationally equivalent* and can be used interchangeably [PCPT14].

At this point we could almost formulate a conjecture such as

$$\begin{array}{c} \Gamma w \dashv | Q_0 \\ \vdots \\ \Gamma v \dashv \end{array}$$

where  $Q_0$  is the process representing the initial state of the machine that transforms input  $w$  to output  $v$ . Before reading on, consider why this may not hold.<sup>1</sup>

---

<sup>1</sup>We did not discuss this find point regarding the operational semantics in lecture.

Yes: the problem is that when  $T$  attempts to send an output symbol to the right, there is no consumer so the process will actually block and the computation will come to a halt. There are at least two ways to solve this problem. One is to make communication *asynchronous* so that output (sending a label to the left of right) can always take place. This has two advantages: (1) it is more realistic from the implementation perspective, and (2) it increases the available parallelism.

Another solution is to create a client that will accept the expected output string. This client could be written in the form of a finite automaton, which we discuss in the next section.

## 5 Finite-State Automata

A (deterministic) finite-state automaton works almost exactly like a subsequential transducer, but it will output only either *acc* or *rej*, not a whole string. This is easy to model:

$$\text{answer} = \oplus\{\text{acc} : \mathbf{1}, \text{rej} : \mathbf{1}\}$$

It generalizes the grammar for strings by allowing two different endmarkers (instead of just \$), and has otherwise no symbols.

Then we would write

$$\begin{aligned} \text{string} &\vdash \text{reject} : \text{answer} \\ \text{reject} &= \text{caseL } (a \Rightarrow \text{reject} \mid b \Rightarrow \text{reject} \mid \$ \Rightarrow \text{R.rej} ; \leftrightarrow) \\ \text{string} &\vdash \text{accept} : \text{answer} \\ \text{accept} &= \text{caseL } (a \Rightarrow \text{accept} \mid b \Rightarrow \text{accept} \mid \$ \Rightarrow \text{R.acc} ; \leftrightarrow) \\ \text{string} &\vdash \_bab\_ : \text{answer} \\ \_bab\_ &= \text{caseL } (a \Rightarrow \text{reject} \\ &\quad \mid \$ \Rightarrow \text{R.rej} ; \leftrightarrow) \\ &\quad \mid b \Rightarrow \text{caseL } (b \Rightarrow \text{reject} \\ &\quad \mid \$ \Rightarrow \text{R.rej} ; \leftrightarrow) \\ &\quad \mid a \Rightarrow \text{caseL } (a \Rightarrow \text{reject} \\ &\quad \mid \$ \Rightarrow \text{R.rej} ; \leftrightarrow) \\ &\quad \mid b \Rightarrow \text{accept})) \end{aligned}$$

We can then test our machine with

$$\begin{aligned} \ulcorner bbab \urcorner &\mid Q_0 \mid \_bab\_ \\ &\vdots \\ &(R.acc ; \leftrightarrow) \end{aligned}$$

Now we can state more generally that

$$\begin{array}{c} \vdash w \sqcap | Q_0 | \sqcup v \sqcup \\ \vdots \\ (\mathsf{R}.acc ; \leftrightarrow) \end{array}$$

if and only if the FST with initial state  $Q_0$  transduces input  $w$  to output  $v$ . The proof of essentially this theorem is sketched in a recent paper [DP16].

The transitions of the transducer here are not exactly in one-to-one correspondence with the steps of proof construction, since the sequence of reading the input and writing the output are usually seen as a single step. Except for this potential minor difference regarding what counts as a step (depending on the precise formulation of the finite-state transducer), automata transitions are modeled precisely a logical inference steps.

In fact, the opposite is also true. If we have a *cut-free proof*

$$\textit{string} \vdash P : \textit{string}$$

then  $P$  will behave like a finite state transducer. The proof is essentially by inversion: Since the proof is cut-free,  $P$  can proceed only by forwarding, receiving from the left, sending to the right, or recursing. From this we can easily construct an FST with the same behavior, again allowing for some minor discrepancies in how steps are counted.

Taken together, this means we have an isomorphism between proofs in subsingleton logic containing only  $\oplus$  and  $1$  and inductively defined types and subsequential finite state transducers. A recent paper [DP16] slightly generalized FSTs so that they encompass finite-state automata as well by allowing multiple distinct endmarkers, as we have done for the representation of string acceptors.

As a last remark, we notice that composition of transducers is logically trivial, namely just cut. If we have

$$\textit{string} \vdash T_1 : \textit{string} \quad \text{and} \quad \textit{string} \vdash T_2 : \textit{string}$$

then

$$\textit{string} \vdash (T_1 \mid T_2) : \textit{string}$$

Here, the two transducers will run in parallel, similarly to our earlier modeling of transducers via ordered inference.  $T_1$  will pass its output to  $T_2$ , which will in turn pass its output to a consumer on the right. We can also just perform cut elimination to obtain a cut-free  $T'$  equivalent to  $(T_1 \mid T_2)$ ,

but a word of caution: in the presence of corecursive (circular) proofs, the usual cut elimination algorithms has to work somewhat differently [FS13]. Nevertheless, it is an illustration how logical tools such as cut elimination can be used in programming languages, this time in program transformation.

For example, we can write a program *flip* that turns a's into b's and vice versa:

```
string ⊢ flip : string
flip = caseL (a ⇒ R.b ; flip
              | b ⇒ R.a ; flip
              | $ ⇒ R.$ ; ↔)
```

Now we can write a program that compresses runs of a's instead of b's (renaming  $Q_0$  to  $\text{compress}_b$  for consistency):

```
string ⊢ compressa : string
string ⊢ compressb : string
compressa = flip | compressb | flip
compressb = Q0
```

Note that in the process configuration

$$\text{flip} \mid \text{compress}_b \mid \text{flip}$$

the three processes will actually work concurrently, passing through the letters *a* and *b* in the form of a pipeline.

## Exercises

**Exercise 1** Write a transducer over the alphabet  $a, b$  which produces  $ab$  for every occurrence of  $ab$  in the input and erases all other symbols.

1. Present it in the form of ordered inference rules.
2. Present it in the form of a well-typed program.

**Exercise 2** Rewrite your parity-computing inference rules from Exercise L2.2 as a transducer, replacing  $\text{eps}$  with the endmarker  $\$$ .

1. Present the transducer in the form of ordered inference rules, for reference. You may freely change your solution of Exercise L2.2 in order to prepare it for part 2.
2. Rewrite it in the form of a well-typed ordered concurrent program.

**Exercise 3** Rewrite the program below as a finite state transducer, expressed as a set of ordered inference rules. Describe the function on strings that  $Q_0$  computes.

$$\begin{aligned} Q_0 = \text{caseL} & (a \Rightarrow Q_1 \\ & | b \Rightarrow Q_2 \\ & | \$ \Rightarrow R.\$ ; \leftrightarrow) \end{aligned}$$

$$\begin{aligned} Q_1 = \text{caseL} & (a \Rightarrow Q_1 \\ & | b \Rightarrow R.b ; Q_2 \\ & | \$ \Rightarrow R.\$ ; \leftrightarrow) \end{aligned}$$

$$\begin{aligned} Q_2 = \text{caseL} & (a \Rightarrow R.a ; Q_1 \\ & | b \Rightarrow Q_2 \\ & | \$ \Rightarrow R.\$ ; \leftrightarrow) \end{aligned}$$

**Exercise 4** Reconsider the transducers for compressing runs of  $b$ 's, given here as a set of ordered inference rules. We present here the version without an explicit final state.

$$\begin{array}{ccc} \frac{a \ q_0}{q_0 \ a} & \frac{b \ q_0}{q_1 \ b} & \frac{\$ \ q_0}{\$} \\ \\ \frac{a \ q_1}{q_0 \ a} & \frac{b \ q_1}{q_1} & \frac{\$ \ q_1}{\$} \end{array}$$

In our encoding as a program  $Q_0$  of type  $\text{string} \vdash Q_0 : \text{string}$  we treated letters as messages and states as processes. No explicit representation of the final state is necessary with the rules above.

Define a dual encoding where symbols of the alphabet and endmarkers are represented processes and states as messages.

1. Define an appropriate type  $\text{state}$  so that  $\text{state} \vdash P_a : \text{state}$  where  $P_a$  is the process representation for the alphabet symbol  $a$ .
2. For each symbol  $a$  of the transducer alphabet, define the process  $P_a$ .
3. Give the type of the process  $P_{\$}$  representing the endmarker  $\$$ . You may choose whether to represent a final state as an explicit message of some form or not.
4. Define the process  $P_{\$}$  for the endmarker.
5. Define the initial configuration for the string  $babb$  and initial state  $q_0$ . Then describe it in general for the machine under consideration here.
6. Define the final configuration for the given example string and initial state. Then describe it in general for the machine under consideration here.
7. Do you foresee any difficulties for encoding subsequential finite state transducers in general in this style? Note that FSTs read one symbol at a time but may output any number of symbols (including none) in one transition. Describe how this could be handled, or explain why a dual construction may only work for a restricted class of FSTs.
8. Consider how to compose transducers and compare to the composition in the original encoding given in lecture.

## References

- [Bae12] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1), 2012.
- [CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.
- [CPT13] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 2013. To appear. Special Issue on Behavioural Types.
- [Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.
- [DP16] Henry DeYoung and Frank Pfenning. Substructural proofs as automata. In *14th Asian Symposium on Programming Languages and Systems*, Hanoi, Vietnam, November 2016. Springer LNCS. To appear.
- [FS13] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: Semantics and cut elimination. In *22nd Conference on Computer Science Logic*, volume 23 of *LIPics*, pages 248–262, 2013.
- [GH05] Simon J. Gay and Malcolm Hole. Subtyping for session types in the  $\pi$ -calculus. *Acta Informatica*, 42(2–3):191–225, 2005.
- [HHN<sup>+</sup>14] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniéou, and Nobuko Yoshida. Structuring communication with session types. In *Concurrent Objects and Beyond (COB 2014)*, pages 105–127. Springer LNCS 8665, 2014.
- [How69] W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.
- [PCPT14] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences

for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

- [TCP14] Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In *Proceedings of the 9th International Symposium on Trustworthy Global Computing (TGC 2014)*, Rome, Italy, September 2014. To appear.
- [Ton15] Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Carnegie Mellon University and New University of Lisbon, 2015. In preparation.
- [Wad12] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming*, ICFP 2012, pages 273–286, Copenhagen, Denmark, September 2012. ACM Press.

# Lecture Notes on Ordered Proofs as Concurrent Programs

15-317: Constructive Logic  
Frank Pfenning

Lecture 24  
November 30, 2017

## 1 Introduction

In this lecture we begin with a summary of the correspondence between proofs and programs for subsingleton logic, carry out some new examples, and then consider how the interpretation might be generalized to the case of ordered logic with more than one antecedent.

## 2 Concurrent Subsingleton Programs

Types	$A ::= \oplus\{l_i : A_i\}_{i \in I}$	internal choice
	$\&\{l_i : A_i\}_{i \in I}$	external choice
	$\mathbf{1}$	termination
Processes	$P, Q ::= \leftrightarrow$	forward
	$  (P \mid Q)$	compose
	$  R.l_k ; P$	send label right
	$  \text{caseL}(l_i \Rightarrow Q_i)_{i \in I}$	receive label left
	$  \text{caseR}(l_i \Rightarrow P_i)_{i \in I}$	receive label right
	$  L.l_k ; Q$	send label left
	$  \text{closeR}$	close and notify right
	$  \text{waitL} ; Q$	wait on close left
		$\mathbf{1R}$
		$\mathbf{1L}$

We also allow mutually recursive type definitions  $\alpha = A$  which must be *contractive*, that is,  $A$  must be of the form  $\oplus\{\dots\}$ ,  $\&\{\dots\}$ , or  $\mathbf{1}$ . We treat a

type name as equal to its definition and will therefore silently replace it. The usual manner of making this more explicit is to use types of the form  $\mu\alpha.A$ , but we forego this exercise here.

Similarly, we allow mutually recursive process definitions of variables  $X$  as processes  $P$  in the form  $\omega \vdash X = P : A$ . Collectively, these constitute the program  $\mathcal{P}$ . We fix a global program  $\mathcal{P}$  so that the typing judgment, formally, is  $\omega \vdash_{\mathcal{P}} P : A$  where we assume that  $\omega \vdash_{\mathcal{P}} Q : A$  for every definition  $\omega \vdash X = Q : A$  in  $\mathcal{P}$ . Since  $\mathcal{P}$  does not change in any typing derivation, we omit this subscript in the rules.

$$\begin{array}{c}
 \frac{}{A \vdash \leftrightarrow : A} \text{id}_A \quad \frac{\omega \vdash P : A \quad A \vdash Q : C}{\omega \vdash (P \mid Q) : C} \text{cut}_A \\
 \frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash (\mathbf{R}.l_k ; P) : \oplus\{l_i : A_i\}_{i \in I}} \oplus R_k \quad \frac{A_i \vdash Q_i : C \quad (\text{for all } i \in I)}{\oplus\{l_i : A_i\}_{i \in I} \vdash \text{caseL}(l_i \Rightarrow Q_i)_{i \in I} : C} \oplus L \\
 \frac{\omega \vdash P_i : A_i \quad (\text{for all } i \in I)}{\omega \vdash \text{caseR}(l_i \Rightarrow P_i)_{i \in I} : \&\{l_i : A_i\}_{i \in I}} \& R \quad \frac{A_k \vdash Q : C \quad (k \in I)}{\&\{l_i : A_i\}_{i \in I} \vdash (\mathbf{L}.l_k ; Q) : C} \& L_k \\
 \frac{\cdot \vdash Q : C}{\cdot \vdash \text{closeR} : \mathbf{1}} \mathbf{1}R \quad \frac{\mathbf{1} \vdash \text{waitL} ; Q : C}{\mathbf{1} \vdash \text{waitL} ; Q : C} \mathbf{1}L \\
 \frac{(\omega \vdash X = P : A) \in \mathcal{P}}{\omega \vdash X : A} X
 \end{array}$$

For the synchronous operational semantics presented via ordered inference, we use ephemeral propositions  $\text{proc}(P)$  which expresses the current state of an executing process  $P$ . We also import the process definitions

$X = P$  as persistent propositions  $\underline{\text{def}}(X, P)$ .

$$\begin{array}{c}
 \frac{\leftrightarrow \text{ fwd}}{\cdot} \quad \frac{(P \mid Q) \text{ cmp}}{P \mid Q} \\
 \frac{(\text{R}.l_k ; P) \mid (\text{caseL}(l_i \Rightarrow Q_i)_{i \in I})}{P \mid Q_k} \oplus C \\
 \frac{(\text{caseR}(l_i \Rightarrow P_i)_{i \in I}) \mid (\text{L}.l_k ; Q)}{P_k \mid Q} \& C \\
 \frac{\text{closeR} \mid (\text{waitL} ; Q)}{Q} \mathbf{1}C \\
 \frac{X \quad \underline{\text{def}}(X, P)}{P} \text{ def}
 \end{array}$$

### 3 Computing with Binary Numbers

We return to the example of numbers in binary notation. A number such as  $(11)_{10} = (1011)_2$  is represented as the ordered context

$$e \cdot b1 \cdot b0 \cdot b1 \cdot b1$$

Increment can be specified using ordered inference by adding an  $\text{inc}$  proposition on the right and the following rules of inference

$$\begin{array}{ccc}
 \frac{b0 \text{ inc}}{b1} & \frac{b1 \text{ inc}}{\text{inc } b0} & \frac{e \text{ inc}}{e \text{ b1}}
 \end{array}$$

For example, we may make the following inferences

$$\begin{array}{c}
 \frac{e \cdot b1 \cdot b0 \cdot b1 \cdot b1 \cdot \text{inc}}{e \cdot b1 \cdot b0 \cdot b1 \cdot \text{inc} \cdot b0} \\
 \frac{e \cdot b1 \cdot b0 \cdot \text{inc} \cdot b0 \cdot b0}{e \cdot b1 \cdot b0 \cdot b0 \cdot b0}
 \end{array}$$

In this lecture we are interested in formulating this kind of computation via message-passing concurrency. This means we have to identify which of the propositions  $e$ ,  $b0$ ,  $b1$  and  $\text{inc}$  are to be viewed as *messages* and which are

to be viewed as *processes*. This choice is not uniquely determined, but can lead to very different styles of programs.

For want of a better name, we will call our two styles *functional* and *object-oriented*. In the functional style,  $e$ ,  $b0$ , and  $b1$  are messages and  $inc$  is a process. In the object-oriented style  $e$ ,  $b0$ , and  $b1$  are processes and  $inc$  is a message.

## 4 Quasi-Functional Increment

In the quasi-functional version,  $inc$  is implemented as a process *increment* that receives a stream of bits from the left representing the number  $n$  and produces a stream of bits on the right representing  $n + 1$ .

First, streams of bits (type *bin*) are represented very similarly to words from the previous lecture.

$$\begin{aligned} bin &= \oplus\{b0 : bin, b1 : bin, e : \mathbf{1}\} \\ bin &\vdash incr : bin \end{aligned}$$

The type of *incr* expresses that it transforms one number into another. The type dictates that *incr* will have to start by reading from the left.

$$\begin{aligned} incr &= \text{caseL} ( b0 \Rightarrow \dots \\ &\quad | b1 \Rightarrow \dots \\ &\quad | e \Rightarrow \dots ) \end{aligned}$$

In the case we receive  $b0$  we have to output  $b1$  *and we are done*. What we mean here by “being done” is that from then on, the remaining input bits are passed on unchanged. We can implement this with an identity process, or with forwarding, where the latter is more efficient and also more concise.

$$\begin{aligned} incr &= \text{caseL} ( b0 \Rightarrow R.b1 ; \leftrightarrow \\ &\quad | b1 \Rightarrow \dots \\ &\quad | e \Rightarrow \dots ) \end{aligned}$$

In the case we receive  $b1$  we have to send  $b0$ , but because of the required carry we still have to incr the remainder of the input stream of bits. This turns into a recursive call to *incr*.

$$\begin{aligned} incr &= \text{caseL} ( b0 \Rightarrow R.b1 ; \leftrightarrow \\ &\quad | b1 \Rightarrow R.b0 ; incr \\ &\quad | e \Rightarrow \dots ) \end{aligned}$$

When the input stream is empty (which represents the integer 0), we have to output the integer 1, which is b1 followed by e. After that, both input and output stream have type **1**, so we can terminate by forwarding.

$$\text{incr} = \text{caseL} (\text{b0} \Rightarrow \text{R.b1} ; \leftrightarrow \\ | \text{b1} \Rightarrow \text{R.b0} ; \text{incr} \\ | \text{e} \Rightarrow \text{R.b1} ; \text{R.e} ; \leftrightarrow)$$

## 5 Quasi-Object-Oriented Increment

The other possibility of interpreting our logical specification as message-passing concurrent computation is to turn e, b0 and b1 into processes, and inc into a message. A number such as

$$\text{e} \cdot \text{b1} \cdot \text{b0} \cdot \text{b1} \cdot \text{b1}$$

then constitutes a configuration of five processes

$$\text{emp} \mid \text{bit1} \mid \text{bit0} \mid \text{bit1} \mid \text{bit1}$$

We can think of each of these processes as an “object”, where we can send a message to the rightmost object only. Because we can only increment it, we think of each of these objects as a counter. At the moment, the only message we can send is an increment message inc, so we have

$$\begin{aligned} \text{counter} &= \&\{\text{inc} : \text{counter}\} \\ &\cdot \vdash \text{emp} : \text{counter} \\ &\text{counter} \vdash \text{bit0} : \text{counter} \\ &\text{counter} \vdash \text{bit1} : \text{counter} \end{aligned}$$

Let’s start with the *bit0* process: it simply absorbs the *inc* message and turns into *bit1*:

$$\text{bit0} = \text{caseR} (\text{inc} \Rightarrow \text{bit1})$$

The *bit1* process has to turn into a *bit0* process, but is also has to send an increment message to its left, representing the carry.

$$\text{bit1} = \text{caseR} (\text{inc} \Rightarrow \text{L.inc} ; \text{bit0})$$

Finally, *emp* does not have to send on any message, but spawn a new process and continue as *bit1*:

$$\text{emp} = \text{caseR} (\text{inc} \Rightarrow \text{empty} \mid \text{bit1})$$

## 6 Converting Between Styles

Here is a summary of the types and code so far.

**Increment in quasi-functional style.**

$$\begin{aligned} bin &= \oplus\{b0 : bin, b1 : bin, e : 1\} \\ bin \vdash incr : bin \\ incr &= \text{caseL} ( b0 \Rightarrow R.b1 ; \leftrightarrow \\ &\quad | b1 \Rightarrow R.b0 ; incr \\ &\quad | e \Rightarrow R.b1 ; R.e ; \leftrightarrow ) \end{aligned}$$

There is concurrency here in that multiple increment processes can be active at the same time, processing the incoming bits in a pipeline.

**A counter in quasi-object-oriented style.**

$$\begin{aligned} counter &= \&\{inc : counter\} \\ \cdot \vdash emp : counter \\ counter \vdash bit0 : counter \\ counter \vdash bit1 : counter \\ bit0 &= \text{caseR} (inc \Rightarrow bit1) \\ bit1 &= \text{caseR} (inc \Rightarrow L.inc ; bit0) \\ emp &= \text{caseR} (inc \Rightarrow empty | bit1) \end{aligned}$$

Here, concurrency is embodied in multiple increment messages being in flight at the same time as they flow through the network of processes from right to left.

**Conversions between representations.** To implement conversions between these representation means to implement to processes that mediate between the types.

$$\begin{aligned} counter \vdash value : bin \\ bin \vdash toctr : counter \end{aligned}$$

**Extracting the value of a counter.** First, *value* which extracts the value from the counter to its left. In order to implement this, we need to add a new kind of message to the *counter* interface, let's call it *val*.

$$\text{counter} = \&\{\text{inc} : \text{counter}, \\ \text{val} : \text{bin}\}$$

We then extend the implementations of *bit0*, *bit1* and *emp* to account for this new kind of message.

$$\begin{aligned} \text{bit0} &= \text{caseR} (\text{inc} \Rightarrow \text{bit1} \\ &\quad | \text{val} \Rightarrow \text{R.b0} ; \text{L.val} ; \leftrightarrow) \\ \text{bit1} &= \text{caseR} (\text{inc} \Rightarrow \text{L.inc} ; \text{bit0} \\ &\quad | \text{val} \Rightarrow \text{R.b1} ; \text{L.val} ; \leftrightarrow) \\ \text{emp} &= \text{caseR} (\text{inc} \Rightarrow \text{empty} | \text{bit1} \\ &\quad | \text{val} \Rightarrow \text{R.e} ; \leftrightarrow) \end{aligned}$$

**Viewing a binary number as a counter.** In our first implementation, a counter exists as a whole sequence of  $\log_2(n + 1)$  processes to hold the number  $n$ , each process holding one bit. In this implementation the counter actually receives a stream of bits to its left and behaves as a counter to its client.

Let's first look at the case when the counter receives an increment message *inc*. In that case we have to spawn a new increment process *incr* to increment the stream of bits coming from the left. These two fit together because

$$\frac{\text{bin} \vdash \text{incr} : \text{bin} \quad \text{bin} \vdash \text{toctr} : \text{counter}}{\text{bin} \vdash (\text{incr} | \text{toctr}) : \text{counter}}$$

$\text{bin} \vdash \text{toctr} : \text{counter}$

$$\text{toctr} = \text{caseR} (\text{inc} \Rightarrow \text{incr} | \text{toctr} \\ | \text{val} \Rightarrow \dots)$$

If the counter receives a *val* message it simply forwards, since it already holds the counter value as a stream to its left.

$\text{bin} \vdash \text{toctr} : \text{counter}$

$$\text{toctr} = \text{caseR} (\text{inc} \Rightarrow \text{incr} | \text{toctr} \\ | \text{val} \Rightarrow \leftrightarrow)$$

## 7 From Subsingleton to Ordered Logic

The style of implementation we have discussed so far works well when the problem allows all the processes to be assembled in a straight line, com-

municating only with its immediate neighbors. But what if we want to implement a tree? Or operate on two streams such as adding two binary numbers?

In order to support more complex topologies of process, we generalize from subsingleton logic to ordered logic. The difference is only that we allow multiple antecedents. This is a big change since we immediately obtain four new connectives: over ( $A / B$ ), under ( $A \setminus B$ ), fuse ( $A \bullet B$ ), and twist ( $A \circ B$ ).

Before we get to their operational meaning, let's reconsider the basic judgment. The first attempt is to generalize from

$$A \vdash P : B$$

to

$$A_1 \cdots A_n \vdash P : B$$

The problem now is: How can  $P$  address  $A_i$  if it wants to send or receive a message from it? For example, several of these types might be internal choice, and  $P$  could receive a label from any of them. In subsingleton logic, there was only (at most) a single process to the left, so this was unambiguous.

We could address this by saying, for example, that  $P$  received from the  $i$ th process, essentially numbering the antecedents. This quickly becomes unwieldy, both in practice and in theory. Or we might say that  $P$  can only communicate with, say,  $A_n$  or  $A_1$ , the extremal processes in the antecedents. However, this appears too restrictive. Instead, we uniquely label each antecedent as well as the succedent<sup>1</sup> with a *channel name*.

$$(x_1:A_1) \cdots (x_n:A_n) \vdash P :: (y:B)$$

We read this as

*Process  $P$  provides a service of type  $B$  along channel  $y$  and uses channels  $x_i$  of type  $A_i$ .*

Since we are still in ordered logic, the order of the antecedents matter, and we will see later in which way. We abbreviate it as  $\Omega \vdash P :: (y:B)$ , overloading  $\Omega$  to stand either for just an ordered sequence of antecedent or one where each antecedent is labeled.

We now generalize each of the rules from before.

---

<sup>1</sup>Not strictly necessary, since the conclusion remains a singleton, but convenient to correlate providers with their clients through a private shared channel.

**Cut.** Instead of simply writing  $P \mid Q$ , the two processes  $P$  and  $Q$  share a private channel.

$$\frac{\Omega \vdash P_x :: (x:A) \quad \Omega_L (x:A) \quad \Omega_R \vdash Q_x :: (z:C)}{\Omega_L \Omega \Omega_R \vdash (x \leftarrow P_x ; Q_x) :: (z:C)} \text{ cut}$$

As a point of notation, we subscript processes variables such as  $P$  or  $Q$  with *bound variables* if they are allowed to occur in them. In the process expression  $x \leftarrow P_x ; Q_x$ , the variable  $x$  is bound and occurs on both side, because it is a channel connecting the two processes. We almost maintain the invariant that all channel names in the antecedent and succedent are distinct, possibly renaming bound variable silently to maintain that.

Operationally, the process executing  $(x \leftarrow P_x ; Q_x)$  continues as  $Q_x$  while spawning a new process  $P_x$ . This interpretation is meaningful since both  $(x \leftarrow P_x ; Q_x)$  and  $Q_x$  offer service  $C$  along  $z$ . This asymmetry in the operational interpretation comes from the asymmetry of ordered logic (and intuitionistic logic in general) with multiple antecedents but at most one succedent.

In order to define the operational semantics, we write  $\text{proc}(x, P)$  if the process  $P$  provides along channel  $x$ , which is to say it is typed as  $\Omega \vdash P :: (x:A)$  for some  $\Omega$  and  $A$ . This is useful to track communications. Then for cut we have the generalized rule of composition

$$\frac{\text{proc}(z, x \leftarrow P_x ; Q_x)}{\text{proc}(w, P_w) \quad \text{proc}(z, Q_w)} \text{ cmp}^w$$

We write  $\text{cmp}^w$  to remind ourselves that the channel  $w$  must be globally “fresh”: it is not allowed to occur anywhere else in the process configuration.

**Identity.** The identity rule could just be

$$\frac{}{y:A \vdash \leftrightarrow :: (x:A)} \text{id}$$

based on the idea that  $x$  and  $y$  are known at this point in a proof so they don’t need to be mentioned. Experience dictates that easily recognizing whenever channels are used makes programs much more readable, so we write

$$\frac{}{y:A \vdash x \leftarrow y :: (x:A)} \text{id}$$

and read is as  $x$  is implemented by  $y$  or  $x$  forwards to  $y$ .

There are various levels of detail in the operational semantics for describing identity in the presence of channel names. We cannot simply terminate the process, but we need to actively connect  $x$  with  $y$ . One way to do this is to globally identify them, which we can do in ordered inference by using equality (which we have not introduced yet).

$$\frac{\text{proc}(x, x \leftarrow y)}{x = y} \text{ fwd}$$

**Internal Choice.** This should be straightforward: instead of sending a label “to the right”, we send it along the channel the process provides.

$$\frac{\Omega \vdash P :: (x:A_k) \quad (k \in I)}{\Omega \vdash (x.l_k ; P) :: (x : \oplus\{l_i:A_i\}_{i \in I})} \oplus R_k$$

Conversely, for the left rule we just receive along a channel of the right type, rather than receiving from the right.

$$\frac{\Omega_L (x:A_i) \Omega_R \vdash Q_i :: (z:C) \quad (\forall i \in I)}{\Omega_L (x:\oplus\{l_i:A_i\}_{i \in I}) \Omega_R \vdash \text{case } x (l_i \Rightarrow Q_i)_{i \in I} :: (z:C)} \oplus L$$

Communication of the label goes through a channel. We only show the synchronous version:

$$\frac{\text{proc}(x, x.l_k ; P) \quad \text{proc}(z, \text{case } x (l_i \Rightarrow Q_i)_{i \in I})}{\text{proc}(x, P) \quad \text{proc}(z, Q_k)} \oplus C$$

The fly in the ointment here is that these two processes may actually not be next to each other, because a client can not be next to all of its providers now that there is more than one.

One possible solution is to send messages (asynchronously) and allow them to be move past other messages and processes. This, however, does not seem a faithful representation of channel behavior, and a single communication could take many steps of exchange. A simpler solution is to retreat to *linear inference* where the order of the propositions no longer matters. We have used this, for example, to describe the spanning tree construction, Hamiltonian cycles, blocks world, etc. Now we reuse it for the operational semantics. Our earlier rules for cut and identity should also be reinterpreted in linear and not ordered inference.

**External Choice.** This is symmetric to internal choice and therefore boring and postponed.

**Unit.** The previous pattern generalizes nicely: instead of closeR and waitL we close and wait on a channel.

$$\frac{}{\cdot \vdash \text{close } x :: (x:\mathbf{1})} \mathbf{1}R \quad \frac{\Omega_L \Omega_R \vdash Q :: (z:C)}{\Omega_L (x:\mathbf{1}) \Omega_R \vdash (\text{wait } x ; Q) :: (z:C)} \mathbf{1}L$$

$$\frac{\text{proc}(x, \text{close } x) \quad \text{proc}(z, \text{wait } x ; Q)}{\text{proc}(z, Q)} \mathbf{1}C$$

**Fuse.** The natural right rule for fuse has two premises.

$$\frac{\Omega_1 \vdash A \quad \Omega_2 \vdash B}{\Omega_1 \Omega_2 \vdash A \bullet B} \bullet R$$

In order to avoid spawning a new process in this rule, we are looking for an sufficient one-premise version. We accomplish this by requiring that either of the two premises must be the identity. So either  $\Omega_1 = A$  or  $\Omega_2 = B$ . These considerations yield:

$$\frac{\Omega \vdash B}{A \Omega \vdash A \bullet B} \bullet R^* \quad \frac{\Omega \vdash A}{\Omega B \vdash A \bullet B} \bullet R^\dagger$$

Rather arbitrarily we pick the first, which yields the following pair of right and left rules for  $A \bullet B$

$$\frac{\Omega \vdash B}{A \Omega \vdash A \bullet B} \bullet R^* \quad \frac{\Omega_L A B \Omega_R \vdash C}{\Omega_L (A \bullet B) \Omega_R \vdash C} \bullet L$$

Again, we can ask which of the rules carries information, and here it is  $\bullet R^*$  which sends. Filling in channel names, we see that once again a channel is sent and received.

$$\frac{\Omega \vdash P :: (x:B)}{(w:A) \Omega \vdash (\text{send } x w ; P) :: (x:A \bullet B)} \bullet R^* \quad \frac{\Omega_L (y:A) (x:B) \Omega_R \vdash Q_y :: (z:C)}{\Omega_L (x:A \bullet B) \Omega_R \vdash (y \leftarrow \text{recv } x ; Q_y) :: (z:C)} \bullet L$$

The computation rule implements the intended operational behavior directly.

$$\frac{\text{proc}(x, \text{send } x w ; P) \quad \text{proc}(z, y \leftarrow \text{recv } x ; Q_y)}{\text{proc}(P) \quad \text{proc}([w/y]Q_y)} \bullet C$$

## 8 Lists

With the constructs we have so far, we can now define the type  $\text{list}_A$  of lists with elements of some arbitrary type  $A$ . An “element” here is actually a channel. For example, a list of binary numbers from earlier in the lecture would be  $\text{list}_{\text{bin}}$ , a list of counters would be  $\text{list}_{\text{counter}}$ .

$$\text{list}_A = \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\}$$

If we are using a channel  $l : \text{list}_A$ , the above type expresses that we either receive a `cons` label followed by a channel of type  $A$  and then another list, or we receive a `nil` label followed by an end message closing the channel.

As an example, we develop a process that takes two lists,  $l$  and  $k$ , and produces the result  $r$  of appending them. On each line we show the state of the type of the channels in the form  $\Omega \vdash (r : C)$  as we fill in the process. We begin by receiving a label from  $l$ , just like the functional code would be a case of the structure of  $l$ .

$$\text{list}_A = \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\}$$

$$(l : \text{list}_A) (k : \text{list}_A) \vdash \text{append} : (r : \text{list}_A)$$

$$\begin{aligned} \text{append} = \text{case } l & ( \text{nil} \Rightarrow \dots \\ & | \text{cons} \Rightarrow \dots ) \end{aligned}$$

When the input list is empty, the result list  $r$  is simply  $k$ , which we accomplish just by forwarding. We just have to make sure to wait for the termination of  $l$ .

$$\text{list}_A = \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\}$$

$$(l : \text{list}_A) (k : \text{list}_A) \vdash \text{append} : (r : \text{list}_A)$$

$$\begin{aligned} \text{append} = \text{case } l & ( \text{nil} \Rightarrow & \% (l : \mathbf{1}) (k : \text{list}_A) \vdash (r : \text{list}_A) \\ & | \text{wait } l ; & \% (k : \text{list}_A) \vdash (r : \text{list}_A) \\ & r \leftarrow k & \% (l : A \bullet \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\ & | \text{cons} \Rightarrow \dots & \end{aligned}$$

In the case for `cons`, we have exposed the underlying ordered pair  $A \bullet \text{list}_A$ . Checking against the left rule for  $\bullet$

$$\frac{\Omega_L (y:A) (x:B) \Omega_R \vdash Q_y :: (z:C)}{\Omega_L (x:A \bullet B) \Omega_R \vdash (y \leftarrow \text{recv } x ; Q_y) :: (z:C)} \bullet L$$

we see we can receive the element and it will be added to the antecedents in order

$$\begin{aligned}
 \text{list}_A &= \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\} \\
 (l : \text{list}_A) (k : \text{list}_A) &\vdash \text{append} : (r : \text{list}_A) \\
 \text{append} &= \text{case } l \ (\text{nil} \Rightarrow \text{wait } l ; r \leftarrow k \\
 &\quad | \text{cons} \Rightarrow \quad \% (l : A \bullet \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\
 &\quad \quad x \leftarrow \text{recv } l \quad \% (x : A) (l : A \bullet \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\
 &\quad \quad \dots)
 \end{aligned}$$

Next, we should send a cons label and then  $x$  along  $r$ : this is how much of the output list  $r$  we already know at this point. And we have to do this because we cannot recurse before the context has the right form again. Checking the form of the  $\bullet R^*$  rule

$$\frac{\Omega \vdash P :: (x:B)}{(w:A) \Omega \vdash (\text{send } x w ; P) :: (x:A \bullet B)} \bullet R^*$$

we see that it is possible to send  $x$  because it is indeed at the left end of the context.

$$\begin{aligned}
 \text{list}_A &= \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\} \\
 (l : \text{list}_A) (k : \text{list}_A) &\vdash \text{append} : (r : \text{list}_A) \\
 \text{append} &= \text{case } l \ (\text{nil} \Rightarrow \text{wait } l ; r \leftarrow k \\
 &\quad | \text{cons} \Rightarrow \quad \% (l : A \bullet \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\
 &\quad \quad x \leftarrow \text{recv } l \quad \% (x : A) (l : \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\
 &\quad \quad r.\text{cons} ; \quad \% (x : A) (l : \text{list}_A) (k : \text{list}_A) \vdash (r : A \bullet \text{list}_A) \\
 &\quad \quad \text{send } r x ; \quad \% (l : \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\
 &\quad \quad \dots)
 \end{aligned}$$

Now we can recurse, completing the program.

$$\begin{aligned}
 \text{list}_A &= \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\} \\
 (l : \text{list}_A) (k : \text{list}_A) &\vdash \text{append} : (r : \text{list}_A) \\
 \text{append} &= \text{case } l \ (\text{nil} \Rightarrow \text{wait } l ; r \leftarrow k \\
 &\quad | \text{cons} \Rightarrow \quad \% (l : A \bullet \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\
 &\quad \quad x \leftarrow \text{recv } l \quad \% (x : A) (l : \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\
 &\quad \quad r.\text{cons} ; \quad \% (x : A) (l : \text{list}_A) (k : \text{list}_A) \vdash (r : A \bullet \text{list}_A) \\
 &\quad \quad \text{send } r x ; \quad \% (l : \text{list}_A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\
 &\quad \quad \text{append})
 \end{aligned}$$

Interestingly, it appears<sup>2</sup> all terminating processes  $P$  with the type

$$(l : \text{list}_A) (k : \text{list}_A) \vdash P : (r : \text{list}_A)$$

---

<sup>2</sup>a conjecture, at present...

will have to append  $l$  and  $k$  in this order, even if the details on how this is accomplished may differ. For example, let's imagine we want to read an element from  $k$  and send this on to  $r$  in a hypothetical process  $exapp$ :

$$\begin{aligned} \text{list}_A &= \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\} \\ (l : \text{list}_A) (k : \text{list}_A) &\vdash exapp : (r : \text{list}_A) \\ exapp &= \text{case } k (\text{nil} \Rightarrow \dots \\ &\quad | \text{cons} \Rightarrow \qquad \qquad \qquad \% (l : \text{list}_A) (k : A \bullet \text{list}_A) \vdash (r : \text{list}_A) \\ &\quad \quad \quad x \leftarrow \text{recv } k \qquad \% (l : \text{list}_A) (x : A) (k : \text{list}_A) \vdash (r : \text{list}_A) \\ &\quad \quad \quad \dots) \end{aligned}$$

At this point we cannot send  $x$  along  $r$  because  $x$  is not at the left end of the context. So the constraints imposed by ordered logic significantly constrain the space of possibly implementations. If we worked in linear logic instead, where the order of hypotheses didn't matter, sending  $x$  here would be possible. The only guarantee we would get<sup>3</sup> is that  $r$  contains all the elements from  $l$  and  $k$  in some arbitrary order.

---

<sup>3</sup>a conjecture, at present...

# Lecture Notes on Stacks and Queues

15-317: Constructive Logic  
Frank Pfenning

Lecture 25  
December 5, 2017

## 1 Introduction

We begin this section by writing some more examples on ordered lists and then implementations of lists and queues in an object-oriented style.

Recall the definition of ordered antecedents

$$\Omega ::= (c : A) \mid \epsilon \mid \Omega_1 \cdot \Omega_2$$

where  $\cdot$  is an associative operator with unit  $\epsilon$ , and  $c$  is interpreted as a channel. The basic judgment

$$(c_1 : A_1) \cdots (c_n : A_n) \vdash P :: (c : A)$$

means that  $P$  is a process *using* channels  $c_i$  and *providing* channel  $c$ .

We summarize the language constructs so far in the following table, from the point of view of the provider of a service.

Type	Provider	Continuation	Client
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c.k$	$c : A_k$	$\text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L}$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L}$	$c : A_k$	$c.k$
$c : \mathbf{1}$	$\text{close } c$	$(\text{none})$	$\text{wait } c ; Q$
$c : A \bullet B$	$\text{send } c d$	$c : B$	$x \leftarrow \text{recv } c ; Q_x$

## 2 List Constructors

Recall the type of ordered lists

$$\text{list}_A = \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\}$$

Even though this session type describes a message interface rather than data layed out in memory, and it is ordered rather than composed of arbitrary pairs, the analogy to data types in functional language should be clear. For example, in the syntax of Standard ML, we might translate this type as

```
datatype 'a list =
  cons of 'a * 'a list
  | nil of unit
```

In a functional language, this give us `cons` and `nil` as constructors. Are there analogous *processes* here? Let's consider

$$(x : A) (l : \text{list}_A) \vdash \text{cons} :: (r : \text{list}_A)$$

When defining `cons`,  $x$ ,  $l$ , and  $r$  are process parameters, which we indicate by

$$r \leftarrow \text{cons} \leftarrow x \ l = \dots$$

where the body of the process definition refers to  $r$ ,  $x$ , and  $l$ . The intent is that  $r$  represents the list with head  $x$  and tail  $l$ . It can announce the fact that it starts with a `cons` by sending this:

$$\begin{aligned} r \leftarrow \text{cons} \leftarrow x \ l = \\ r.\text{cons} ; & \quad \% (x : A) (l : \text{list}_A) \vdash r : A \bullet \text{list}_A \\ \dots \end{aligned}$$

Now we have to send a channel of type  $A$  along  $r$ . Fortunately, the element  $x : A$  is at the left end of the antecedents so the ordering restriction on  $\bullet R^*$  is satisfied and we can send it.

$$\begin{aligned} r \leftarrow \text{cons} \leftarrow x \ l = \\ r.\text{cons} ; & \quad \% (x : A) (l : \text{list}_A) \vdash r : A \bullet \text{list}_A \\ \text{send } r \ x ; & \quad \% (l : \text{list}_A) \vdash r : \text{list}_A \\ \dots \end{aligned}$$

Now that we have sent  $x$ , we express that the remainder of the list  $r$  is  $l$  by forwarding  $l$  to  $r$ . Fortunately, the types are arranged in the right way.

$$\begin{aligned} r \leftarrow \text{cons} \leftarrow x \ l = \\ r.\text{cons} ; & \quad \% (x : A) (l : \text{list}_A) \vdash r : A \bullet \text{list}_A \\ \text{send } r \ x ; & \quad \% (l : \text{list}_A) \vdash r : \text{list}_A \\ r \leftarrow l \end{aligned}$$

The *nil* constructor for a process representing the empty list works analogously.

$$\begin{aligned} \vdash \textit{nil} :: (r : \text{list}_A) \\ r \leftarrow \textit{nil} = r.\text{nil} ; \text{close } r \end{aligned}$$

### 3 Following the Types

Next, we explore the prescriptive power of types. Which mystery processes would have type

$$(l : \text{list}_A) (x : A) \vdash \textit{myst} :: (r : \text{list}_A)$$

This is almost the type of *cons*, except that the order of the antecedents is reversed. If we were working in linear logic, where  $A \bullet B$  is symmetric, *cons* would in fact satisfy this type, but not in ordered logic because after sending the *cons* label

$$\begin{aligned} r \leftarrow \textit{myst} \leftarrow l \ x = \\ r.\text{cons} ; & \quad \% (l : \text{list}_A) (x : A) \vdash r : A \bullet \text{list}_A \\ \dots \end{aligned}$$

we cannot send *x* because it is not at the left end of the antecedents.

Intuitively, if the list *l* is “virtually” in the context with its elements in order, then maybe we should be able to add *x* at the end. This means we actually have to read the elements from *l*, similar to our implementation of *append*.

$$\begin{aligned} (l : \text{list}_A) (x : A) \vdash \textit{myst} :: (r : \text{list}_A) \\ r \leftarrow \textit{myst} \leftarrow l \ x = \\ \text{case } l (\text{cons} \Rightarrow y \leftarrow \text{recv } l ; & \quad \% (y : A) (l : \text{list}_A) (x : A) \vdash (r : \text{list}_A) \\ \dots) \end{aligned}$$

At this point we can send *cons* and then *y* along *r*, but not *x*, which will only be available once the list *l* has been transferred to *r* in its entirety. Once we have sent *y*, we can recurse, passing the same *l* and *x* back to *myst*.

$$\begin{aligned} (l : \text{list}_A) (x : A) \vdash \textit{myst} :: (r : \text{list}_A) \\ r \leftarrow \textit{myst} \leftarrow l \ x = \\ \text{case } l (\text{cons} \Rightarrow y \leftarrow \text{recv } l ; & \quad \% (y : A) (l : \text{list}_A) (x : A) \vdash (r : \text{list}_A) \\ r.\text{cons} ; \text{send } r \ y ; & \quad \% (l : \text{list}_A) (x : A) \vdash r : \text{list}_A \\ r \leftarrow \textit{myst} \leftarrow l \ x \\ | \text{ nil} \Rightarrow \dots) \end{aligned}$$

In the case of *nil* we can wait for  $l$  to close and then send *cons*,  $x$ , *nil*, and then close.

$$\begin{aligned}
 & (l : \text{list}_A) (x : A) \vdash \text{myst} :: (r : \text{list}_A) \\
 & r \leftarrow \text{myst} \leftarrow l \ x = \\
 & \quad \text{case } l \ (\text{cons} \Rightarrow y \leftarrow \text{recv } l ; \quad \% (y : A) (l : \text{list}_A) (x : A) \vdash (r : \text{list}_A) \\
 & \quad \quad r.\text{cons} ; \text{send } r \ y ; \quad \% (l : \text{list}_A) (x : A) \vdash r : \text{list}_A \\
 & \quad \quad r \leftarrow \text{myst} \leftarrow l \ x \\
 & \quad | \ \text{nil} \Rightarrow \quad \text{wait } l ; \quad \% (x : A) \vdash r : \text{list}_A \\
 & \quad \quad r.\text{cons} ; \text{send } r \ x \quad \% \vdash r : \text{list}_A \\
 & \quad \quad r.\text{nil} ; \text{close } r)
 \end{aligned}$$

We can also use our *nil* and *cons* in several places instead falling back on sending messages directly.

$$\begin{aligned}
 & (l : \text{list}_A) (x : A) \vdash \text{myst} :: (r : \text{list}_A) \\
 & r \leftarrow \text{myst} \leftarrow l \ x = \\
 & \quad \text{case } l \ (\text{cons} \Rightarrow y \leftarrow \text{recv } l ; \quad \% (y : A) (l : \text{list}_A) (x : A) \vdash r : \text{list}_A \\
 & \quad \quad s \leftarrow \text{myst} \leftarrow l \ x ; \quad \% (y : A) (s : \text{list}_A) \vdash r : \text{list}_A \\
 & \quad \quad r \leftarrow \text{cons} \leftarrow y \ s \\
 & \quad | \ \text{nil} \Rightarrow \quad \text{wait } l ; \quad \% (x : A) \vdash r : \text{list}_A \\
 & \quad \quad n \leftarrow \text{nil} ; \quad \% (x : A) (n : \text{list}_A) \vdash r : \text{list}_A \\
 & \quad \quad r \leftarrow \text{cons} \leftarrow x \ n)
 \end{aligned}$$

Note how, for example, the recursive call to *myst* consumes  $(l : \text{list}_A) (x : A)$  from the antecedents and replace it by  $(s : \text{list}_A)$ . This stems from the form of the ordered cut rule

$$\frac{\Omega \vdash P :: (x : A) \quad \Omega_L \cdot (x : A) \cdot \Omega_R \vdash Q :: (z : C)}{\Omega_L \cdot \Omega \cdot \Omega_R \vdash (x \leftarrow P ; Q) :: (z : C)} \text{ cut}$$

where here  $\Omega_L = (y : A)$ ,  $\Omega = (l : \text{list}_A) (x : A)$ , and  $\Omega_R = \epsilon$ .

Also, in the case of *nil* where we spawn a new *nil* process, no antecedents are consumed, so the new channel  $n$  can go anywhere in the context. We will need it to the right of  $x : A$  so we can call *cons* next to construct a singleton list.

So we see there are different programs, with different behavior (for example, the placement of the recursive call), and yet in the end the observable behavior along  $r$  should be the same: the elements of  $l$  followed by  $x$ .

## 4 Ordered Implications

Now we move on from the quasi-functional style a quasi-object-oriented example: implementing a store with insert and delete operations. Rather than an internal choice ( $\oplus$ ) like lists, the interface will present the client with a choice between insertion and deletion in the form of an external choice ( $\&$ ).

Here is our simple interface to a storage service for channels:

$$\text{store}_A = \& \{ \text{ins} : A \setminus \text{store}_A, \\ \text{del} : \oplus \{ \text{none} : \mathbf{1}, \text{some} : A \bullet \text{store}_A \} \}$$

In the first line we see that the provider may receive and `ins` message, and then it must receive the channel to insert into the store. We model this with  $A \setminus B$ , which we have not yet introduced. First, the right rule:

$$\frac{(y:A) \Omega \vdash P_y :: (x : B)}{\Omega \vdash (y \leftarrow \text{recv } x ; P_y) :: (x : A \setminus B)} \setminus R$$

Operationally, a provider of  $x : A \setminus B$  receives a channel of type  $A$ , adds it to the left end of the antecedents, and continues with  $x : B$ .

A client process using a channel  $x : A \setminus B$  must therefore send a channel of type  $A$ . Moreover, due to ordering constraints, this channel must be immediately to the left of  $x$ .

$$\frac{\Omega_L (x : B) \Omega_R \vdash Q :: (z : C)}{\Omega_L (w : A) (x : A \setminus B) \Omega_R \vdash (\text{send } x w ; Q) :: (z : C)} \setminus L^*$$

The following computation rule implements the cut reduction of  $\setminus R$  and  $\setminus L^*$ .

$$\frac{\text{proc}(x, y \leftarrow \text{recv } x ; P_y) \quad \text{proc}(z, \text{send } x w ; Q)}{\text{proc}(x, [w/y]P_y) \quad \text{proc}(z, Q)} \setminus C$$

Note that the operational reading here is *identical* for  $B / A$ ; the difference is entirely in the restrictions about where  $w:B$  or  $y:B$  are to be found. In the  $/R$  rule it will be added to the right of the antecedents and in the  $/L^*$  rule it must be immediately to the right of the receiving channel.

$$\frac{\Omega (y:A) \vdash P_y :: (x : B)}{\Omega \vdash (y \leftarrow \text{recv } x ; P_y) :: (x : B / A)} / R$$

$$\frac{\Omega_L (x : B) \Omega_R \vdash Q :: (z : C)}{\Omega_L (x : B / A) (w : A) \Omega_R \vdash (\text{send } x w ; Q) :: (z : C)} / L^*$$

## 5 Implementing a Store

Recall the desired type of a store interface, using  $A \setminus B$ :

$$\text{store}_A = \&\{ \text{ins} : A \setminus \text{store}_A, \\ \text{del} : \oplus\{\text{none} : \mathbf{1}, \text{some} : A \bullet \text{store}_A\} \}$$

Using our operational interpretation, we can read this as follows:

*A store for channels of type A offers a client a choice between insertion (label ins) and deletion (label del).*

*When inserting, the clients sends a channel of type A which is added to the store.*

*When deleting, the store responds with the label none if there are no elements in the store and terminates, or with the label some, followed by an element.*

*When an element is actually inserted or deleted the provider of the storage service then waits for the next input (again, either an insertion or deletion).*

In this reading we have focused on the operations, and intentionally ignored the restrictions order might place on the use of the storage service. Hopefully, this will emerge as we write the code and analyze what the restrictions might mean.

First, we have to be able to create an empty store. We will write the code in stages, because I believe it is much harder to understand the final program than it is to follow its construction.

$$\text{store}_A = \&\{ \text{ins} : A \setminus \text{store}_A, \\ \text{del} : \oplus\{\text{none} : \mathbf{1}, \text{some} : A \bullet \text{store}_A\} \}$$

First, the header of the process definition.

$$\begin{aligned} & \cdot \vdash \text{empty} :: (s : \text{store}_A) \\ & s \leftarrow \text{empty} = \dots \end{aligned}$$

Because a  $\text{store}_A$  is an external choice, we begin with a case construct, branching on the received label.

$$\begin{aligned} & \cdot \vdash \text{empty} :: (s : \text{store}_A) \\ & s \leftarrow \text{empty} = \text{case } s \text{ (} \text{ins} \Rightarrow & \% \quad \cdot \vdash s : A \setminus \text{store}_A \\ & \quad | \text{del} \Rightarrow \dots & \% \quad \cdot \vdash s : \oplus\{\text{none} : \mathbf{1}, \text{some} : A \bullet \text{store}_A\} \\ & \quad ) \end{aligned}$$

The case of deletion is actually easier: since this process represents an empty store, we send the label `none` and terminate.

$$\begin{array}{c} \cdot \vdash \text{empty} :: (s : \text{store}_A) \\ s \leftarrow \text{empty} = \text{case } s \text{ (ins} \Rightarrow \quad \% \quad \cdot \vdash s : A \setminus \text{store}_A \\ \quad | \text{ del} \Rightarrow s.\text{none} ; \text{close } s) \end{array}$$

In the case of an insertion, the type dictates that we receive a channel of type  $A$  which we call  $x$ . It is added at the left end of the antecedents. Since they are actually none, both  $A \setminus \text{store}_A$  and  $\text{store}_A / A$  would behave the same way here.

$$\begin{array}{c} \cdot \vdash \text{empty} :: (s : \text{store}_A) \\ s \leftarrow \text{empty} = \text{case } s \text{ (ins} \Rightarrow \quad \% \quad \cdot \vdash s : A \setminus \text{store}_A \\ \quad | x \leftarrow \text{recv } s ; \% \quad x : A \vdash s : \text{store}_A \\ \quad | \dots \\ \quad | \text{ del} \Rightarrow s.\text{none} ; \text{close } s) \end{array}$$

At this point it seems like we are stuck. We need to start a process implementing a store with *one* element, but so far we just writing the code for an empty store. We need to define a process *elem*

$$(x:A) (t:\text{store}_A) \vdash \text{elem} :: (s : \text{store}_A)$$

which holds an element  $x:A$  and also another store  $t:\text{store}_A$  with further elements. In the singleton case,  $t$  will then be the empty store. Therefore, we first make a recursive call to create another empty store, calling it  $n$  for *none*.

$$\begin{array}{c} \cdot \vdash \text{empty} :: (s : \text{store}_A) \\ s \leftarrow \text{empty} = \text{case } s \text{ (ins} \Rightarrow x \leftarrow \text{recv } s ; \quad \% \quad x : A \vdash s : \text{store}_A \\ \quad | n \leftarrow \text{empty} ; \quad \% \quad (x : A) (n : \text{store}_A) \vdash s : \text{store}_A \\ \quad | \dots \\ \quad | \text{ del} \Rightarrow s.\text{none} ; \text{close } s) \\ \\ (x : A) (t : \text{store}_A) \vdash \text{elem} :: (s : \text{store}_A) \\ s \leftarrow \text{elem} \leftarrow x \ t = \dots \end{array}$$

Postponing the definition of *elem* for now, we can invoke *elem* to create a singleton store with just  $x$ , calling the resulting channel  $e$ . This call will consume  $x$  and  $n$ , leaving  $e$  as the only antecedent.

$$\cdot \vdash \text{empty} :: (s : \text{store}_A)$$

$$\begin{aligned}
 s \leftarrow empty &= \text{case } s \ (\text{ins} \Rightarrow x \leftarrow \text{recv } s ; & \% \quad x:A \vdash s : \text{store}_A \\
 &\quad n \leftarrow empty ; & \% \quad (x:A) \ (n:\text{store}_A) \vdash s : \text{store}_A \\
 &\quad e \leftarrow elem \leftarrow x \ n ; & \% \quad e:\text{store}_A \vdash s : \text{store}_A \\
 &\quad \dots \\
 &\quad | \ \text{del} \Rightarrow s.\text{none} ; \ \text{close } s) \\
 (x:A) \ (t:\text{store}_A) \vdash elem &:: (s : \text{store}_A) \\
 s \leftarrow elem \leftarrow x \ t &= \dots
 \end{aligned}$$

At this point we can implement  $s$  by  $e$  (the singleton store), which is just an application of the identity rule.

$$\begin{aligned}
 \cdot \vdash empty &:: (s : \text{store}_A) \\
 s \leftarrow empty &= \text{case } s \ (\text{ins} \Rightarrow x \leftarrow \text{recv } s ; & \% \quad (x:A) \vdash s : \text{store}_A \\
 &\quad n \leftarrow empty ; & \% \quad (x:A) \ (n:\text{store}_A) \vdash s : \text{store}_A \\
 &\quad e \leftarrow elem \leftarrow x \ n & \% \quad e:\text{store}_A \vdash s : \text{store}_A \\
 &\quad s \leftarrow e \\
 &\quad | \ \text{del} \Rightarrow s.\text{none} ; \ \text{close } s) \\
 (x:A) \ (t:\text{store}_A) \vdash elem &:: (s : \text{store}_A) \\
 s \leftarrow elem \leftarrow x \ t &= \dots
 \end{aligned}$$

It remains to write the code for the process holding an element of the store. We suggest you reconstruct or at least read it line by line the way we developed the definition of  $empty$ , but we will not break it out explicitly into multiple steps. However, we will still give the types after each interaction. For easy reference, we repeat the type definition for  $\text{store}_A$ .

$$\begin{aligned}
 \text{store}_A = \&\{ \text{ins} : A \setminus \text{store}_A, \\
 &\text{del} : \oplus\{\text{none} : \mathbf{1}, \text{some} : A \bullet \text{store}_A\} \} \\
 (x:A) \ (t:\text{store}_A) \vdash elem &:: (s : \text{store}_A) \\
 1 \quad s \leftarrow elem \leftarrow x \ t &= \\
 2 \quad \text{case } s \ (\text{ins} \Rightarrow y \leftarrow \text{recv } s ; & \% \quad (y:A) \ (x:A) \ (t:\text{store}_A) \vdash s : \text{store}_A \\
 3 \quad \quad t.\text{ins} ; & \% \quad (y:A) \ (x:A) \ (t:A \setminus \text{store}_A) \vdash s : \text{store}_A \\
 4 \quad \quad \text{send } t \ x ; & \% \quad (y:A) \ (t:\text{store}_A) \vdash s : \text{store}_A \\
 5 \quad \quad r \leftarrow elem \leftarrow y \ t ; & \% \quad r:\text{store}_A \vdash s : \text{store}_A \\
 6 \quad \quad s \leftarrow r & \\
 7 \quad \quad | \ \text{del} \Rightarrow s.\text{some} ; & \% \quad (x:A) \ (t:\text{store}_A) \vdash s : A \bullet \text{store}_A \\
 8 \quad \quad \text{send } s \ x ; & \% \quad t:\text{store}_A \vdash s : \text{store}_A \\
 9 \quad \quad s \leftarrow t &
 \end{aligned}$$

A few notes on this code. Look at the type at the end of the *previous* line to understand the next line.

- In line 2, we add  $y:A$  at the left end of the context since  $s : A \setminus \text{store}_A$ .
- In line 4, we can only pass  $x$  to  $t$  but not  $y$ , due restrictions of  $\setminus L^*$ .
- In line 5,  $y$  and  $t$  are in the correct order to call  $\text{elem}$  recursively.
- In line 8, we can pass  $x$  along  $s$  since it is at the left end of the context.

How does this code behave? Assume we have a store  $s$  holding elements  $x_1$  and  $x_2$  it would look like

$\text{proc}(s, s \leftarrow \text{elem} \leftarrow x_1 t_1) \quad \text{proc}(t_1, t_1 \leftarrow \text{elem} \leftarrow x_2 t_2) \quad \text{proc}(t_2, t_2 \leftarrow \text{empty})$

where we have indicated the code executing in each process without unfolding the definition. If we insert an element along  $s$  (by sending  $\text{ins}$  and then a new  $y$ ) then the process  $s \leftarrow \text{elem} \leftarrow x_1 t_1$  will insert  $x_1$  along  $t_1$  and then, in two steps, become  $s \leftarrow \text{elem} \leftarrow y t_1$ . Now the next process will pass  $x_2$  along  $t_2$  and hold on to  $x_1$ , and finally the process holding no element will spawn a new one ( $t_3$ ) and itself hold on to  $x_2$ .

$\text{proc}(s, s \leftarrow \text{elem} \leftarrow y t_1) \quad \text{proc}(t_1, t_1 \leftarrow \text{elem} \leftarrow x_1 t_2)$   
 $\quad \quad \quad \text{proc}(t_2, t_2 \leftarrow \text{elem} \leftarrow x_2 t_3) \quad \text{proc}(t_3, t_3 \leftarrow \text{empty})$

If we next delete an element, we will get  $y$  back and the store will effectively revert to its original state, with some (internal) renaming.

$\text{proc}(s, s \leftarrow \text{elem} \leftarrow x_1 t_2) \quad \text{proc}(t_2, t_2 \leftarrow \text{elem} \leftarrow x_2 t_3) \quad \text{proc}(t_3, t_3 \leftarrow \text{empty})$

In essence, the store behaves like a *stack*: the most recent element we have inserted will be the first one deleted. If you carefully look through the intermediate types in the  $\text{elem}$  process, it seems that this behavior is forced. We conjecture that any implementation of the store interface we have given will behave like a stack or might at some point not respond to further messages.

## 6 Tail Calls

If we look at lines 5 and 6

$$\begin{array}{ll} r \leftarrow \text{elem} \leftarrow y t ; & \% \quad r:\text{store}_A \vdash s : \text{store}_A \\ s \leftarrow r & \end{array}$$

we are starting a new process, providing along a new channel  $r$  and then forward this to  $s$ . Instead, we can simply continue in the current process, executing  $elem$ , which is written as

$$s \leftarrow elem \leftarrow y t ;$$

The examples now simplify very slightly.

$$\begin{aligned} \text{store}_A = & \& \{ \text{ins} : A \setminus \text{store}_A, \\ & \text{del} : \oplus\{\text{none} : \mathbf{1}, \text{some} : A \bullet \text{store}_A\} \} \\ \cdot \vdash empty :: (s : \text{store}_A) \\ s \leftarrow empty = & \text{case } s (\text{ins} \Rightarrow x \leftarrow \text{recv } s ; \quad \% (x:A) \vdash s : \text{store}_A \\ & n \leftarrow empty ; \quad \% (x:A) (n:\text{store}_A) \vdash s : \text{store}_A \\ & s \leftarrow elem \leftarrow x n \quad \% e:\text{store}_A \vdash s : \text{store}_A \\ & | \text{del} \Rightarrow s.\text{none} ; \text{close } s) \\ (x:A) (t:\text{store}_A) \vdash elem :: (s : \text{store}_A) \\ s \leftarrow elem \leftarrow x t = & \text{case } s (\text{ins} \Rightarrow y \leftarrow \text{recv } s ; \quad \% (y:A) (x:A) (t:\text{store}_A) \vdash s : \text{store}_A \\ & t.\text{ins} ; \quad \% (y:A) (x:A) (t:A \setminus \text{store}_A) \vdash s : \text{store}_A \\ & \text{send } t x ; \quad \% (y:A) (t:\text{store}_A) \vdash s : \text{store}_A \\ & s \leftarrow elem \leftarrow y t \\ & | \text{del} \Rightarrow s.\text{some} ; \quad \% (x:A) (t:\text{store}_A) \vdash s : A \bullet \text{store}_A \\ & \text{send } s x ; \quad \% t:\text{store}_A \vdash s : \text{store}_A \\ & s \leftarrow t) \end{aligned}$$

## 7 Queues

As notes, our implementation so far ended up behaving like a stack, and we conjectured that the type of the interface itself forced this behavior. Can we modify the type to allow (and perhaps force) the behavior of the store as a queue, where the first element we store is the first one we receive back? I encourage you to try to work this out before reading on ...

The key idea is to change the type

$$\text{store}_A = \& \{ \text{ins} : A \setminus \text{store}_A, \\ \text{del} : \oplus \{ \text{none} : \mathbf{1}, \text{some} : A \bullet \text{store}_A \} \}$$

to

$$\text{queue}_A = \& \{ \text{ins} : \text{queue}_A / A, \\ \text{del} : \oplus \{ \text{none} : \mathbf{1}, \text{some} : A \bullet \text{queue}_A \} \}$$

We will not go through this in detail, but reading the following code and the type after each interaction should give you a sense for what this change entails.

$$\begin{aligned} & \cdot \vdash \text{empty} :: (s : \text{queue}_A) \\ 1 & \quad s \leftarrow \text{empty} = \\ 2 & \quad \text{case } s (\text{ins} \Rightarrow x \leftarrow \text{recv } s ; \quad \% \quad x:A \vdash s : \text{queue}_A \\ 3 & \quad \quad \quad n \leftarrow \text{empty} ; \quad \% \quad (x:A) (n:\text{queue}_A) \vdash s : \text{queue}_A \\ 4 & \quad \quad \quad s \leftarrow \text{elem} \leftarrow x n \\ 5 & \quad \quad \quad | \text{ del} \Rightarrow s.\text{none} ; \text{close } s) \\ \\ & (x:A) (t:\text{queue}_A) \vdash \text{elem} :: (s : \text{queue}_A) \\ 6 & \quad s \leftarrow \text{elem} \leftarrow x t = \\ 7 & \quad \text{case } s (\text{ins} \Rightarrow y \leftarrow \text{recv } s ; \quad \% \quad (x:A) (t:\text{queue}_A) (y:A) \vdash s : \text{queue}_A \\ 8 & \quad \quad \quad t.\text{ins} ; \quad \% \quad (x:A) (t:\text{queue}_A / A) (y:A) \vdash s : \text{queue}_A \\ 9 & \quad \quad \quad \text{send } t y ; \quad \% \quad (x:A) (t:\text{queue}_A) \vdash s : \text{queue}_A \\ 10 & \quad \quad \quad s \leftarrow \text{elem} \leftarrow x t \\ 11 & \quad \quad \quad | \text{ del} \Rightarrow s.\text{some} ; \quad \% \quad (x:A) (t:\text{queue}_A) \vdash s : A \bullet \text{queue}_A \\ 12 & \quad \quad \quad \text{send } s x ; \quad \% \quad t:\text{queue}_A \vdash s : \text{queue}_A \\ 13 & \quad \quad \quad s \leftarrow t) \end{aligned}$$

The critical changes are in line 7 (where  $y$  is added to the *right end* of the antecedents instead of the left) and line 9 (where consequently  $y$  instead of  $x$  must be sent along  $t$ ).

The complexity of all the operations remains the same, since the only difference is whether the current  $x$  or the new  $y$  is sent along  $t$ , but the implementation now behaves like a queue rather than a stack.

## 8 Summary, and Linear Logic

We complete the table from the beginning of the lecture that summarizes the computational interpretation of ordered logic.

Type	Provider	Continuation	Client
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c.k$	$c : A_k$	$\text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L}$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L}$	$c : A_k$	$c.k$
$c : \mathbf{1}$	$\text{close } c$	(none)	$\text{wait } c ; Q$
$c : A \bullet B$	$\text{send } c d$	$c : B$	$x \leftarrow \text{recv } c ; Q_x$
$c : A \setminus B$	$x \leftarrow \text{recv } c ; P_x$	$c : B$	$\text{send } c d$
$c : B / A$	$x \leftarrow \text{recv } c ; P_x$	$c : B$	$\text{send } c d$

The difference between the last two rows are the places of  $x$  and  $d$  among the antecedents.

In the case of linear logic, the last two lines collapse, using  $A \multimap B$  in a unified notation, and the multiplicative conjunction  $A \bullet B$  no longer is subject to any ordering constraint and is written as  $A \otimes B$ . The programming constructs do not change.

Type	Provider	Continuation	Client
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c.k$	$c : A_k$	$\text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L}$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L}$	$c : A_k$	$c.k$
$c : \mathbf{1}$	$\text{close } c$	(none)	$\text{wait } c ; Q$
$c : A \otimes B$	$\text{send } c d$	$c : B$	$x \leftarrow \text{recv } c ; Q_x$
$c : A \multimap B$	$x \leftarrow \text{recv } c ; P_x$	$c : B$	$\text{send } c d$

# Constructive Logic (15-317), Fall 2017

## Recitation 10: Focused Proof Search

November 1, 2017

### 1 Focused Proof Search

In this recitation we will focus<sup>1</sup> on an alternative proof search strategy for the negative fragment of intuitionistic propositional logic. Previously we have sought to reduce the uncertainty by making the logic more and more restricted without decreasing its power. This has taken us from natural deduction, to the sequent calculus, to a version of the sequent calculus where we apply inversion rules first and finally, now, to focusing. The idea is similar to that behind the inversion strategy, occasionally in the logic we must guess but there are many rules which are always safe to apply. As before we will greedily apply those rules but, unlike before, we will also consolidate our guesses. That is, whenever we're forced to make a guess we make as many guesses as possible. By batching guesses like this, we avoid even more nondeterminism.

The inference rules for this logic are as follows. First the rules that handle inversion. Since we're dealing with negative propositions, they all are all right rules and constitute all the right rules in the system.

$$\frac{\Gamma, A \rightarrow B}{\Gamma \rightarrow A \supset B} \supset R \quad \frac{\Gamma \rightarrow A \quad \Gamma \rightarrow B}{\Gamma \rightarrow A \wedge B} \wedge R \quad \frac{}{\Gamma \rightarrow \top} \top R$$

Next comes the rule for selecting a proposition to focus on.

$$\frac{A \in \Gamma \quad \Gamma; [A] \rightarrow P}{\Gamma \rightarrow P} \text{ focus}$$

Then we have the batched rules for guesses.

$$\frac{\Gamma \rightarrow A \quad \Gamma; [B] \rightarrow P}{\Gamma; [A \supset B] \rightarrow P} \supset L \quad \frac{\Gamma; [A] \rightarrow P}{\Gamma; [A \wedge B] \rightarrow P} \wedge L1 \quad \frac{\Gamma; [B] \rightarrow P}{\Gamma; [A \wedge B] \rightarrow P} \wedge L2 \quad \frac{}{\Gamma; [P] \rightarrow P} \wedge P$$

Notice in particular, the lack of rules for  $P \neq Q$  when focusing on  $Q$ . This means that if focusing on a goal does not produce the result, we backtrack and focus on a new proposition. Now for some examples.

---

<sup>1</sup>Heh.

$$1. ((P \wedge Q) \wedge R) \implies (P \wedge (Q \wedge R))$$

$$\frac{\frac{\frac{\frac{\frac{\overline{(P \wedge Q) \wedge R; [P] \longrightarrow P}}{(P \wedge Q) \wedge R; [P \wedge Q] \longrightarrow P} \quad \frac{\overline{(P \wedge Q) \wedge R; [P \wedge Q] \longrightarrow Q}}{(P \wedge Q) \wedge R; [(P \wedge Q) \wedge R] \longrightarrow Q} \quad \frac{\overline{(P \wedge Q) \wedge R; [R] \longrightarrow R}}{(P \wedge Q) \wedge R; [(P \wedge Q) \wedge R] \longrightarrow P}}{(P \wedge Q) \wedge R \longrightarrow Q} \quad \frac{(P \wedge Q) \wedge R \longrightarrow R}{(P \wedge Q) \wedge R \longrightarrow Q \wedge R}}{(P \wedge Q) \wedge R \longrightarrow P} \quad \frac{(P \wedge Q) \wedge R \longrightarrow Q \wedge R}{(P \wedge Q) \wedge R \longrightarrow Q \wedge R}$$

$$\frac{(P \wedge Q) \wedge R \longrightarrow P \wedge (Q \wedge R)}{\cdot \longrightarrow ((P \wedge Q) \wedge R) \implies (P \wedge (Q \wedge R))}$$

$$2. (P \supset Q \supset R) \supset (P \supset Q) \supset P \supset R$$

$$\frac{\frac{\frac{\frac{\frac{\overline{(P \supset Q \supset R), (P \supset Q), P; [P] \longrightarrow P}}{(P \supset Q \supset R), (P \supset Q), P \longrightarrow P} \quad \frac{\overline{(P \supset Q \supset R), (P \supset Q), P; [Q] \longrightarrow Q}}{(P \supset Q \supset R), (P \supset Q), P; [P \supset Q] \longrightarrow Q}}{(P \supset Q \supset R), (P \supset Q), P \longrightarrow Q} \quad \frac{\overline{(P \supset Q \supset R), (P \supset Q), P; [R] \longrightarrow R}}{(P \supset Q \supset R), (P \supset Q), P; [Q \supset R] \longrightarrow R}}{(P \supset Q \supset R), (P \supset Q), P; [P \supset Q \supset R] \longrightarrow R} \quad \frac{\overline{(P \supset Q \supset R), (P \supset Q), P \longrightarrow R}}{(P \supset Q \supset R), (P \supset Q) \supset P \supset R}}{(P \supset Q \supset R) \longrightarrow (P \supset Q) \supset P \supset R}$$

$$\frac{(P \supset Q \supset R) \longrightarrow (P \supset Q) \supset P \supset R}{\cdot \longrightarrow (P \supset Q \supset R) \supset (P \supset Q) \supset P \supset R}$$

$$3. (P \supset Q \supset R) \supset (P \wedge Q) \supset R$$

$$\frac{\frac{\frac{\frac{\overline{(P \supset Q \supset R), (P \wedge Q); [P] \longrightarrow P}}{(P \supset Q \supset R), (P \wedge Q); [P \wedge Q] \longrightarrow P} \quad \frac{\overline{(P \supset Q \supset R), (P \wedge Q); [P \wedge Q] \longrightarrow Q}}{(P \supset Q \supset R), (P \wedge Q) \longrightarrow Q} \quad \frac{\overline{(P \supset Q \supset R), (P \wedge Q); [R] \longrightarrow R}}{(P \supset Q \supset R), (P \wedge Q); [Q \supset R] \longrightarrow R}}{(P \supset Q \supset R), (P \wedge Q) \longrightarrow P \wedge Q} \quad \frac{(P \supset Q \supset R), (P \wedge Q); [P \supset Q \supset R] \longrightarrow R}{\frac{(P \supset Q \supset R), (P \wedge Q) \longrightarrow R}{\frac{(P \supset Q \supset R), (P \wedge Q) \longrightarrow R}{\frac{(P \supset Q \supset R) \longrightarrow (P \wedge Q) \supset R}{\cdot \longrightarrow (P \supset Q \supset R) \supset (P \wedge Q) \supset R}}}}$$

$$4. ((P \wedge Q) \supset R) \supset P \supset (Q \supset R)$$

$$\frac{\frac{\frac{\overline{((P \wedge Q) \supset R), P, Q; [P] \longrightarrow P}}{((P \wedge Q) \supset R), P, Q \longrightarrow P} \quad \frac{\overline{((P \wedge Q) \supset R), P, Q; [Q] \longrightarrow Q}}{((P \wedge Q) \supset R), P, Q \longrightarrow Q}}{((P \wedge Q) \supset R), P, Q \longrightarrow P \wedge Q} \quad \frac{\overline{((P \wedge Q) \supset R), P, Q; [R] \longrightarrow R}}{((P \wedge Q) \supset R), P, Q; [(P \wedge Q) \supset R] \longrightarrow R}}{((P \wedge Q) \supset R), P, Q; [(P \wedge Q) \supset R] \longrightarrow R} \quad \frac{\overline{((P \wedge Q) \supset R), P, Q \longrightarrow R}}{((P \wedge Q) \supset R), P, Q \longrightarrow R}$$

$$\frac{\overline{((P \wedge Q) \supset R), P \longrightarrow (Q \supset R)}}{((P \wedge Q) \supset R) \longrightarrow P \supset (Q \supset R)}$$

$$\frac{((P \wedge Q) \supset R) \longrightarrow P \supset (Q \supset R)}{\cdot \longrightarrow ((P \wedge Q) \supset R) \supset P \supset (Q \supset R)}$$

## RECITATION 11: FULL FOCUSING, Q&A

RYAN KAVANAGH

We will spend the first half of recitation revisiting focusing. The second part of the recitation will be dedicated to answering questions about tomorrow's midterm<sup>1</sup>.

### 1. FOCUSING

We begin by reminding ourselves about polarities. A *negative proposition* is one with an invertible right rule. A *positive proposition* is one with an invertible left rule. See the handout for a refresher on all of the rules.

We begin by exploring how changing the polarity of atoms can drastically change the shape of a proof. Consider the proposition  $a \vee b \wedge c \supset (a \vee b) \wedge (a \vee c)$ . Suppose we first made every atomic proposition positive:<sup>2</sup>

$$a^+ \vee (b^+ \wedge c^+) \supset \uparrow((a^+ \vee b^+) \wedge (a^+ \vee c^+)).$$

For compactness on the board (and on this page), we will write  $X^+$  for  $(a^+ \vee b^+) \wedge (a^+ \vee c^+)$ . The proof would then look like this:

$$\begin{array}{c}
 \dfrac{}{a^+ \longrightarrow [a^+] \quad a^+ \longrightarrow [a^+] \quad a^+ \longrightarrow [a^+ \vee b^+] \quad a^+ \longrightarrow [a^+ \vee c^+]} \\
 \hline
 a^+ \longrightarrow [X^+]
 \end{array}
 \qquad
 \begin{array}{c}
 \dfrac{b^+, c^+ \longrightarrow [b^+] \quad b^+, c^+ \longrightarrow [c^+]}{b^+, c^+ \longrightarrow [a^+ \vee b^+] \quad b^+, c^+ \longrightarrow [a^+ \vee c^+]} \\
 \hline
 b^+, c^+ \longrightarrow [X^+]
 \end{array}
 \qquad
 \begin{array}{c}
 \dfrac{}{b^+, c^+ \longrightarrow X^+} \\
 \hline
 b^+, c^+ ; \bullet \xrightarrow{L} X^+
 \end{array}
 \qquad
 \begin{array}{c}
 \dfrac{}{c^+ ; b^+ \xrightarrow{L} X^+} \\
 \hline
 \bullet ; b^+ \cdot c^+ \xrightarrow{L} X^+
 \end{array}
 \qquad
 \begin{array}{c}
 \dfrac{\bullet ; a^+ \vee (b^+ \wedge c^+) \xrightarrow{L} X^+}{\bullet ; a^+ \vee (b^+ \wedge c^+) \xrightarrow{R} \uparrow X^+} \\
 \hline
 \bullet ; \bullet \xrightarrow{R} a^+ \vee (b^+ \wedge c^+) \supset \uparrow X^+
 \end{array}$$

Now suppose we made every atomic proposition negative:

$$\downarrow a^- \vee \downarrow (b^- \wedge c^-) \supset \uparrow(\downarrow a^- \vee \downarrow b^-) \wedge \uparrow(\downarrow a^- \vee \downarrow c^-).$$

---

*Date:* 8 November 2017.

<sup>1</sup>Remind students that they have a midterm tomorrow.

<sup>2</sup>Consider having students place the arrows in these to make sure they understand what's going on.

Then we would get the following very different proof.

$$\begin{array}{c}
 \frac{}{[\mathbf{a}^-] \longrightarrow \mathbf{a}^-} \\
 \frac{\mathbf{a}^- \longrightarrow \mathbf{a}^-}{\mathbf{a}^- ; \bullet \xrightarrow{R} \mathbf{a}^-} \\
 \frac{\mathbf{a}^- \longrightarrow [\mathbf{a}^-]}{\mathbf{a}^- \longrightarrow [\mathbf{a}^- \vee \mathbf{b}^-]} \\
 \frac{\mathbf{a}^- \longrightarrow \mathbf{b}^- \wedge \mathbf{c}^-}{\mathbf{a}^- ; \bullet \xrightarrow{L} \mathbf{b}^- \wedge \mathbf{c}^-} \\
 \frac{\mathbf{a}^- \longrightarrow \mathbf{b}^- \wedge \mathbf{c}^-}{\bullet ; \mathbf{a}^- \xrightarrow{L} \mathbf{b}^- \wedge \mathbf{c}^-} \\
 \frac{}{[\mathbf{b}^-] \longrightarrow \mathbf{b}^-} \\
 \frac{}{[\mathbf{b}^- \wedge \mathbf{c}^-] \longrightarrow \mathbf{b}^-} \\
 \frac{\mathbf{b}^- \wedge \mathbf{c}^- \longrightarrow \mathbf{b}^-}{\mathbf{b}^- \wedge \mathbf{c}^- ; \bullet \xrightarrow{R} \mathbf{b}^-} \\
 \frac{\mathbf{b}^- \wedge \mathbf{c}^- \longrightarrow [\mathbf{b}^-]}{\mathbf{b}^- \wedge \mathbf{c}^- \longrightarrow [\mathbf{a}^- \vee \mathbf{b}^-]} \\
 \frac{\mathbf{b}^- \wedge \mathbf{c}^- \longrightarrow [\mathbf{a}^- \vee \mathbf{b}^-]}{\mathbf{b}^- \wedge \mathbf{c}^- \longrightarrow \downarrow \mathbf{a}^- \vee \downarrow \mathbf{b}^-} \\
 \frac{\mathbf{b}^- \wedge \mathbf{c}^- \longrightarrow \downarrow \mathbf{a}^- \vee \downarrow \mathbf{b}^-}{\mathbf{b}^- \wedge \mathbf{c}^- ; \bullet \xrightarrow{L} \downarrow \mathbf{a}^- \vee \downarrow \mathbf{b}^-} \\
 \frac{\bullet ; \mathbf{a}^- \xrightarrow{L} \downarrow \mathbf{a}^- \vee \downarrow \mathbf{b}^-}{\bullet ; \mathbf{b}^- \wedge \mathbf{c}^- \xrightarrow{L} \downarrow \mathbf{a}^- \vee \downarrow \mathbf{b}^-} \\
 \frac{\bullet ; \mathbf{b}^- \wedge \mathbf{c}^- \xrightarrow{R} \uparrow(\mathbf{a}^- \vee \mathbf{b}^-)}{\bullet ; \mathbf{b}^- \wedge \mathbf{c}^- \xrightarrow{R} \uparrow(\mathbf{a}^- \vee \mathbf{b}^-) \wedge \uparrow(\mathbf{a}^- \vee \mathbf{c}^-)} \\
 \frac{\bullet ; \mathbf{b}^- \wedge \mathbf{c}^- \xrightarrow{R} \uparrow(\mathbf{a}^- \vee \mathbf{b}^-) \wedge \uparrow(\mathbf{a}^- \vee \mathbf{c}^-)}{\bullet ; \bullet \xrightarrow{R} \mathbf{a}^- \vee \mathbf{b}^- \wedge \mathbf{c}^- \supset \uparrow(\mathbf{a}^- \vee \mathbf{b}^-) \wedge \uparrow(\mathbf{a}^- \vee \mathbf{c}^-)}
 \end{array}$$

*symmetric case*

If students would like to do examples with atomic propositions of different polarities, do so.<sup>3</sup>

We now turn to an example involving forward chaining<sup>4</sup> and then find the corresponding derived rules of inference. Consider the proposition

$$\mathbf{a} \supset ((\mathbf{a} \vee \mathbf{b}) \supset \mathbf{c}) \supset (\mathbf{c} \supset \mathbf{d}) \supset \mathbf{d}.$$

After applying right inversion sufficiently many times, one reaches

$$\mathbf{a}, (\mathbf{a} \vee \mathbf{b}) \supset \mathbf{c}, \mathbf{c} \supset \mathbf{d} \xrightarrow{R} \mathbf{d}.$$

We make all of the atoms positive to set ourselves up for forward chaining:

$$\mathbf{a}^+, (\mathbf{a}^+ \vee \mathbf{b}^+) \supset \uparrow \mathbf{c}^+, \mathbf{c}^+ \supset \uparrow \mathbf{d}^+ \xrightarrow{R} \mathbf{d}^+.$$

---

<sup>3</sup>This will, however, cut into the Q&A time for reviewing midterm material.

<sup>4</sup>We work in the forward chaining fragment of Lecture 18; this may be confusing after seeing all of the focusing rules. If the students prefer, amend the examples below to use those rules instead.

The proof then looks like follows, where  $\Gamma_0 = \{a^+, (a^+ \vee b^+) \supset \uparrow c^+, c^+ \supset \uparrow d^+\}$ .

$$\begin{array}{c}
 \frac{}{\Gamma_0, c^+, d^+ \xrightarrow{f} [d^+]}
 \\ \frac{}{\Gamma_0, c^+, d^+ \xrightarrow{f} d^+}
 \\ \frac{}{\Gamma_0, c^+ \xrightarrow{f} [c^+] \quad \Gamma_0, c^+, [\uparrow d^+] \xrightarrow{f} d^+}
 \\ \frac{a^+ \in \Gamma_0}{\Gamma_0 \xrightarrow{f} [a^+]}
 \quad \frac{\Gamma_0, c^+, [c^+ \supset \uparrow d^+] \xrightarrow{f} d^+}{\Gamma_0, c^+ \xrightarrow{f} d^+}
 \\ \frac{\Gamma_0 \xrightarrow{f} [a^+ \vee b^+]}{\Gamma_0, [(a^+ \vee b^+) \supset \uparrow c^+] \xrightarrow{f} d^+}
 \\ \frac{}{\Gamma_0 \xrightarrow{f} d^+}
 \end{array}$$

To find the derived rules of inference, we remark that we can only focus on two propositions in  $\Gamma_0$ , namely,  $(a^+ \vee b^+) \supset \uparrow c^+$  and  $c^+ \supset \uparrow d^+$ . Let's focus on these in turn. Remark that we always go from stable sequents to stable sequents in deriving rules of inference.

$$\begin{array}{c}
 \frac{a^+ \in \Gamma}{\Gamma \xrightarrow{f} [a^+]}
 \quad \frac{\Gamma, c^+ \xrightarrow{f} \rho}{\Gamma, [\uparrow c^+] \xrightarrow{f} \rho}
 \\ \frac{\Gamma \xrightarrow{f} [a^+ \vee b^+] \quad \Gamma, [\uparrow c^+] \xrightarrow{f} \rho}{\Gamma, [(a^+ \vee b^+) \supset \uparrow c^+] \xrightarrow{f} \rho}
 \end{array}$$

We then get the derived rule

$$\frac{\Gamma, a^+, c^+ \xrightarrow{f} \rho}{\Gamma, a^+ \xrightarrow{f} \rho} R_0$$

By symmetry, we also get

$$\frac{\Gamma, b^+, c^+ \xrightarrow{f} \rho}{\Gamma, b^+ \xrightarrow{f} \rho} R'_0$$

Focusing on  $c^+ \supset \uparrow d^+$  gives us:

$$\begin{array}{c}
 \frac{c^+ \in \Gamma}{\Gamma \xrightarrow{f} [c^+]}
 \quad \frac{\Gamma, d^+ \xrightarrow{f} \rho}{\Gamma, [\uparrow d^+] \xrightarrow{f} \rho}
 \\ \frac{}{\Gamma, [c^+ \supset \uparrow d^+] \xrightarrow{f} \rho}
 \end{array}$$

which gives the rule

$$\frac{\Gamma, c^+, d^+ \xrightarrow{f} \rho}{\Gamma, c^+ \xrightarrow{f} \rho} R_1$$

Finally, we can focus on the succedent:

$$\frac{d^+ \in \Gamma}{\frac{\Gamma \xrightarrow{f} [d^+]}{\Gamma \xrightarrow{f} d^+}}$$

to get the derived rule

$$\frac{}{\Gamma, d^+ \xrightarrow{f} d^+} R_2$$

The original proof can then be simplified to

$$\frac{\frac{\frac{\Gamma_0, c^+, d^+ \xrightarrow{f} d^+}{\Gamma_0, c^+ \xrightarrow{f} d^+} R_1}{\Gamma_0 \xrightarrow{f} d^+} R_0}{\Gamma_0 \xrightarrow{f} d^+} R_2$$

## 2. MIDTERM Q&A

Answer questions students may have about the material.

# Constructive Logic (15-317), Fall 2017

## Recitation 12: Substructural logics, a substandard\*recap.

### What is a Substructural Logic

The core idea with substructural deductions as was shown in lecture yesterday is that restricting the logic gives us the ability to express new things. Specifically, we got rid of two so called “structural principles”. These are the rules of weakening and exchange:

$$\frac{\mathcal{J}_0 \quad \mathcal{J}_1}{\mathcal{J}_0} \text{ Weakening} \quad \frac{\mathcal{J}}{\mathcal{J} \quad \mathcal{J}} \text{ Contraction} \quad \frac{\mathcal{J}_0 \quad \mathcal{J}_1}{\mathcal{J}_1 \quad \mathcal{J}_0} \text{ Exchange}$$

By getting rid of weakening and contraction our propositions represent not simply eternally true pieces of knowledge but rather ephemeral truths. This opens the possibility of expressing things that are only temporarily true directly in the logic. For instance, it's rather unnatural to encode currency in normal (structural) logic. After all, knowing that you have one dollar entitles you to conclude that you have two dollars! This is an interesting concept but experience has demonstrated that it is not so. If we sacrifice just weakening, we are working with *linear* logic and if we sacrifice both weakening and exchange we are working in an *ordered* logic.

### Binary Numbers

Let us turn first to ordered logic. There's a natural way to encode a binary number in a logic where judgments may not rearrange themselves or suddenly be dropped; we simply turn 01110 into the knowing b0 b1 b1 b1 b0. This would never work in a structural or even linear logic (why?). For the sake of convenience, we actually denote the end of a binary number with \$. So we would properly encode the above number as

\$ b0 b1 b1 b1 b1 b0

Note that in this encoding we explicitly don't care about preserving standard form. The least significant bit is on the right and the most significant bit is on the left but leading zeroes are explicitly allowed for simplicity.

**Task 1.** Design a proposition inc so that when inc is placed to the right of a binary number it is possible to derive the successor of that binary number.

---

\*GET IT?!?

$$\frac{\$ \quad \text{inc}}{\$ \quad \text{b1}} \quad \frac{\text{b0} \quad \text{inc}}{\text{b1}} \quad \frac{\text{b1} \quad \text{inc}}{\text{inc} \quad \text{b0}}$$

**Task 2.** Design a proposition  $\text{dec}$  so that when  $\text{dec}$  is placed to the right of a binary number it is possible to derive the successor of that binary number. If there is no positive predecessor of the number derive  $\circledast$ .

$$\frac{\$ \quad \text{dec}}{\circledast} \quad \frac{\circledast \quad \text{b1}}{\circledast} \quad \frac{\text{b0} \quad \text{dec}}{\text{dec} \quad \text{b1}} \quad \frac{\text{b1} \quad \text{inc}}{\text{b0}}$$

A worthwhile exercise is to modify these programs so that they do expect and preserve standard forms of binary numbers.

## Graphs

Moving to a new example, let us consider how we might encode graphs in this framework. Ordered logic is a less natural fit here so we turn instead to linear logic. Let us suppose that we have a number of propositions  $\text{node}(a)$  and  $\text{edge}(a, b)$  denoting the obvious things. We shall assume that the graph is undirected but it will be ideal if we do not include both  $\text{edge}(a, b)$  and  $\text{edge}(b, a)$  for every edge. What can we express now with inference?

**Task 3.** Write a series of inference rules to deduce when there is a path from  $a$  to  $b$ .

$$\frac{\text{edge}(a, b)}{\text{edge}(b, a)} \quad \frac{\text{node}(a)}{\text{path}(a, a)} \quad \frac{\text{path}(a, b) \quad \text{edge}(b, c)}{\text{path}(a, c)} \quad \frac{\text{edge}(a, b) \quad \text{path}(b, c)}{\text{path}(a, c)}$$

By encoding these substructurally we have prevented ourselves from being able to derive a path that uses the same edge multiple times. This means that all paths derived may not be the shortest paths, but they are at least *locally minimal*.

As a final example, let us attempt to count the number of paths between two nodes comprised of disjoint paths.

**Task 4.** Write a proposition which counts the number of disjoint paths from  $a$  to  $b$  so that  $\text{connected}(a, b, n)$  is derivable if and only if there are at least  $n$  completely disjoint paths from  $a$  to  $b$ . Assume that at the beginning of the derivation in addition to all of the propositions representing a graph that you are supplied with  $\text{start}$ .

$$\frac{\text{start}}{\text{connected}(a, b, z)} \quad \frac{\text{connected}(a, b, n) \quad \text{path}(a, b)}{\text{connected}(a, b, s(n))}$$

## RECITATION 13: SUBSINGLETON LOGIC

RYAN KAVANAGH

Yesterday, we saw how we could view ordered proofs as concurrent programs. We focused on the subsingleton fragment of ordered logic, namely, the fragment where each judgment has at most one antecedent. We introduced the judgment  $\omega \vdash P : A$ , which under the proofs-as-programs interpretation, can equivalently be viewed as saying that  $P$  is a proof of  $A$  using  $\omega$ , and that  $P$  is a process providing  $A$  and using  $\omega$ . Finally, we saw that the computational interpretation comes from cut reduction.

### 1. RULES OF SUBSINGLETON LOGIC

The subsingleton fragment is given by the following rules, where  $\omega$  is an ordered context consisting of zero or one antecedents:

$$\begin{array}{c}
 \frac{A \vdash C \quad B \vdash C}{A \oplus B \vdash C} \oplus L \quad \frac{\omega \vdash A}{\omega \vdash A \oplus B} \oplus R_1 \quad \frac{\omega \vdash B}{\omega \vdash A \oplus B} \oplus R_2 \\
 \\ 
 \frac{A \vdash C}{A \& B \vdash C} \& L_1 \quad \frac{B \vdash C}{A \& B \vdash C} \& L_2 \quad \frac{\omega \vdash A \quad \omega \vdash B}{\omega \vdash A \& B} \& R \\
 \\ 
 \frac{}{A \vdash A} id_A \quad \frac{\omega \vdash A \quad A \vdash C}{\omega \vdash C} cut_A \quad \frac{\cdot \vdash C}{1 \vdash C} 1L \quad \frac{\cdot \vdash 1}{\cdot \vdash 1} 1R
 \end{array}$$

where we had the following term assignments:

$$\begin{array}{c}
 \frac{A_i \vdash Q_i : C \quad (\forall i \in I)}{\oplus_{\{l_i : A_i\}_{i \in I}} \vdash \text{casel } (l_i \Rightarrow Q_i)_{i \in I} : C} \oplus L \quad \frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash R.l_k; P : \oplus_{\{l_i : A_i\}_{i \in I}} \oplus R_k} \oplus R_k \\
 \\ 
 \frac{\cdot \vdash C}{1 \vdash \text{waitL} : C} 1L \quad \frac{\cdot \vdash \text{closeR} : 1}{\cdot \vdash \text{closeR} : 1} 1R \\
 \\ 
 \frac{}{A \vdash \leftrightarrow : A} id_A \quad \frac{\omega \vdash P : A \quad A \vdash Q : C}{\omega \vdash (P \mid Q) : C} cut_A
 \end{array}$$

(we didn't assign terms to external choice and they won't be used below, but they are symmetric relative to the  $\vdash$ ). These were united by the following reduction rules:

$$\frac{(R.l_k; P) \mid \text{casel } (l_i \Rightarrow Q_i)_{i \in I}}{P \mid Q_k} \quad \frac{\text{closeR} \mid (\text{waitL}; Q)}{Q} \quad \frac{\leftrightarrow \mid Q}{Q} \quad \frac{Q \mid \leftrightarrow}{Q}$$

We remark that we treat  $\mid$  associatively, that is to say, that  $P \mid (Q \mid R)$  and  $(P \mid Q) \mid R$  have the same reductions.

## 2. PROGRAMMING WITH SUBSINGLETON LOGIC

We now consider three examples. We adopt the convention that “;” binds more tightly than “|”, that’s to say, we interpret  $P; Q | R; S$  as  $(P; Q) | (R; S)$  instead of  $P; (Q | R); S$ .

**2.1. Producing strings.** We first want to write a program that produces a given (constant) string. We assume the type  $\text{string}$  to be given by  $\text{str} = \oplus\{\text{a} : \text{str}, \text{b} : \text{str}, \$ : \mathbf{1}\}$ . Then  $\mathbf{1} \vdash \lceil \text{ababbb} \rceil : \text{str}$  is given by

$$\lceil \text{ababbb} \rceil = \text{R.a}; \text{R.b}; \text{R.a}; \text{R.b}; \text{R.b}; \text{R.b}; \text{R.\$}; \leftrightarrow.$$

What would have happened had we not included  $\leftrightarrow$  at the end? We would end up with something ill-typed (cf. the  $\oplus R_k$  rule above). For the sake of illustration, we show the typing derivation of  $\mathbf{1} \vdash \lceil \text{a} \rceil : \text{str}$ :

$$\frac{\overline{\mathbf{1} \vdash \leftrightarrow : \mathbf{1}} \quad \text{id}_{\mathbf{1}}}{\mathbf{1} \vdash \text{R.\$}; \leftrightarrow : \text{str}} \oplus R_{\$}$$

$$\frac{\mathbf{1} \vdash \text{R.\$}; \leftrightarrow : \text{str}}{\mathbf{1} \vdash \text{R.a}; \text{R.\$}; \leftrightarrow : \text{str}} \oplus R_a$$

**2.2. Incrementing Binary Integers.** We define the type  $\text{bin} = \oplus\{\text{b1} : \text{bin}, \text{b0} : \text{bin}, \$ : \mathbf{1}\}$ , analogously to  $\text{str}$ , and encode binary numbers  $b_0 \dots b_n$  in exactly the same way as we encoded strings, except that we reverse the bits (we do so due to the cut rule, as will be made clear below). For example, 110 is encoded as  $\lceil 110 \rceil = \text{R.b0}; \text{R.b1}; \text{R.b1}; \text{R.\$}; \leftrightarrow$ .

Given a binary number string  $b_0 \dots b_n$ , give a program  $\text{inc}$  of type  $\text{bin} \vdash \text{inc} : \text{bin}$  that increments it. In particular,  $\lceil b_0 \dots b_n \rceil | \text{inc}$  should produce something of type  $\text{bin}$  to the right that corresponds to the increment of  $b_0 \dots b_n$ . It is useful to consider how we would implement this in ordered logic:

$$\frac{\text{b0} \quad \text{inc}}{\text{b1}} \quad \frac{\text{b1} \quad \text{inc}}{\text{inc} \quad \text{b0}} \quad \frac{\$ \quad \text{inc}}{\$ \quad \text{b1}}$$

The corresponding program is

$$\text{inc} = \text{caseL} (\text{b0} \Rightarrow \text{R.b1}; \leftrightarrow | \text{b1} \Rightarrow \text{R.b0}; \text{inc} | \$ \Rightarrow \text{R.b1}; \text{R.\$}; \leftrightarrow)$$

We consider an example evaluation. For illustration purposes, we consider the helper program  $\text{turkey}$  that gobbles up everything passed to it from the left:

$$\text{turkey} = \text{caseL} (\text{b0} \Rightarrow \text{turkey} | \text{b1} \Rightarrow \text{turkey} | \$ \Rightarrow \leftrightarrow).$$

Then we see that  $\text{turkey}$  does indeed gobble the increment  $\lceil 10 \rceil$  of 1:

$$\frac{\text{R.b1}; \text{R.\$}; \leftrightarrow | \text{inc} | \text{turkey}}{\text{R.\$}; \leftrightarrow | \text{R.b0}; \text{inc} | \text{turkey}}$$

$$\frac{\text{R.\$}; \leftrightarrow | \text{inc} | \text{turkey}}{\text{R.b1}; \text{R.\$}; \leftrightarrow | \text{turkey}}$$

$$\frac{\text{R.b1}; \text{R.\$}; \leftrightarrow | \text{turkey}}{\text{R.\$}; \leftrightarrow | \text{turkey}}$$

$$\frac{\text{R.\$}; \leftrightarrow | \text{turkey}}{\leftrightarrow | \leftrightarrow}$$

(For illustrative purposes, the input consumed by  $\text{turkey}$  is in red.)

Instead of *turkey*, we could instead have defined an id process:

$$\text{id} = \text{caseL } (\text{b0} \Rightarrow (\text{id} \mid \text{R.b0}; \leftrightarrow) \mid \text{b1} \Rightarrow (\text{id} \mid \text{R.b1}) \mid \$ \Rightarrow \text{R.\$}; \leftrightarrow).$$

This process acts as the identity process, and when computing examples, has the advantage of leaving its input in sight:

$$\begin{array}{c} \text{R.b1; R.\$; } \leftrightarrow \mid \text{inc} \mid \text{id} \\ \hline \text{R.\$; } \leftrightarrow \mid \text{R.b0; inc} \mid \text{id} \\ \hline \text{R.\$; } \leftrightarrow \mid \text{inc} \mid \text{id} \mid \text{R.b0}; \leftrightarrow \\ \hline \text{R.b1; R.\$; } \leftrightarrow \mid \text{id} \mid \text{R.b0}; \leftrightarrow \\ \hline \text{R.\$; } \leftrightarrow \mid \text{id} \mid \text{R.b1}; \leftrightarrow \mid \text{R.b0}; \leftrightarrow \\ \hline \leftrightarrow \mid \text{R.\$; } \leftrightarrow \mid \text{R.b1}; \leftrightarrow \mid \text{R.b0}; \leftrightarrow \\ \hline \text{R.\$; } \leftrightarrow \mid \text{R.b1}; \leftrightarrow \mid \text{R.b0}; \leftrightarrow \end{array}$$

Again, we see that we end up with a process that will produce the intended output  $\lceil 10 \rceil$ .

**2.3. String Reversal.** Given some string  $c_0 \cdots c_n$ , give a program `rev` of type  $\text{str} \vdash \text{rev} : \text{str}$  that provides its reversal to the right. Assume the encoding given above.

We introduce the following thunks as helper functions, where  $k \in \{a, b\}$ :

$$T_k = \text{caseL } (\$ \Rightarrow \text{R.k}; \text{R.\$}; \leftrightarrow \mid a \Rightarrow \text{R.a}; T_k \mid b \Rightarrow \text{R.b}; T_k).$$

Then  $\text{str} \vdash T_k : \text{str}$  acts as the identity on all inputs from the left, except for  $\text{R.\$}$ , on which it outputs the thunked value  $\text{R.k}$  followed by  $\text{R.\$}$ , and then terminates. We can now capture  $\text{str} \vdash \text{rev} : \text{str}$  as follows:

$$\text{rev} = \text{caseL } (a \Rightarrow (\text{rev} \mid_{\text{str}} T_a) \mid b \Rightarrow (\text{rev} \mid_{\text{str}} T_b) \mid \$ \Rightarrow \text{R.\$}; \leftrightarrow).$$

We illustrate this with  $\lceil ab \rceil$ , again using (an analogous) id to gobble up the output and marking the value passed to the right in red.

$$\begin{array}{c} \lceil ab \rceil \mid \text{rev} \mid \text{id} \\ \hline \text{R.a; R.b; R.\$; } \leftrightarrow \mid \text{rev} \mid \text{id} \\ \hline \text{R.b; R.\$; } \leftrightarrow \mid \text{rev} \mid T_a \mid \text{id} \\ \hline \text{R.\$; } \leftrightarrow \mid \text{rev} \mid T_b \mid T_a \mid \text{id} \\ \hline \leftrightarrow \mid \text{R.\$; } \leftrightarrow \mid T_b \mid T_a \mid \text{id} \\ \hline \text{R.\$; } \leftrightarrow \mid T_b \mid T_a \mid \text{id} \\ \hline \leftrightarrow \mid \text{R.b; R.\$; } \leftrightarrow \mid T_a \mid \text{id} \\ \hline \text{R.b; R.\$; } \leftrightarrow \mid T_a \mid \text{id} \\ \hline \text{R.\$; } \leftrightarrow \mid \text{R.b; T_a \mid id \mid R.b; } \leftrightarrow \\ \hline \text{R.\$; } \leftrightarrow \mid T_a \mid \text{id} \mid \text{R.b; } \leftrightarrow \\ \hline \leftrightarrow \mid \text{R.a; R.\$; } \leftrightarrow \mid \text{id} \mid \text{R.b; } \leftrightarrow \\ \hline \text{R.a; R.\$; } \leftrightarrow \mid \text{id} \mid \text{R.a; } \leftrightarrow \mid \text{R.b; } \leftrightarrow \\ \hline \text{R.\$; } \leftrightarrow \mid \text{id} \mid \text{R.a; } \leftrightarrow \mid \text{R.b; } \leftrightarrow \\ \hline \leftrightarrow \mid \text{R.\$; } \leftrightarrow \mid \text{R.a; } \leftrightarrow \mid \text{R.b; } \leftrightarrow \\ \hline \text{R.\$; } \leftrightarrow \mid \text{R.a; } \leftrightarrow \mid \text{R.b; } \leftrightarrow \end{array}$$

Then, any process composed with the above to the right will first see  $R.b$ , then  $R.a$ , and finally  $R.\$$ , i.e., the reversal of  $\lceil ab \rceil$ .

## RECITATION 14: ORDERED PROGRAMMING

RYAN KAVANAGH

Please note that the final exam will be on Tuesday, December 12, 5:30pm–8:30pm, in BH A51. This exam is closed book, closed notes. Tomorrow's lecture will be dedicated to reviewing material for the final exam. There will also be office hours at their regularly slated times this Friday, Saturday, and Monday. Please make use of these resources!

We also posted links to course FCEs and the TA evaluation forms on piazza. Please fill these out: it is the only feedback we get.

Homework 10 was due yesterday, but if you still have grace days leftover, you have until 13:30 tomorrow to hand it in. We will do our best to finish grading it by Friday so that you can get review our feedback before the final. We will make a piazza post once it is graded, and you will be able to collect it during our office hours.

### 1. ORDERED LOGIC TERM ASSIGNMENT RULES

We review the rules assigning terms to ordered logic. In subsingleton logic, we had that contexts were generated by  $\omega ::= \cdot \mid A$  and the judgment  $\omega \vdash P : B$ . We generalize this to the judgment

$$\Omega \vdash P :: (c : A)$$

where  $\Omega$  is an ordered context generated by the grammar  $\Omega ::= \cdot \mid c : A \mid \Omega_1 \cdot \Omega_2$ . We read the judgment

$$(d_1 : A_1) \cdots (d_n : A_n) \vdash P :: (c : A)$$

as meaning the process  $P$  provides a service of type  $A$  along channel  $c$ , and uses channels  $d_i$  of type  $A_i$ . To emphasise that  $d_1, \dots, d_n$  and  $c$  are bound in  $P$ , we will instead notate process declarations as:

$$c \leftarrow P \leftarrow d_1 \dots d_n = P(c, d_1, \dots, d_n).$$

The rule driving computation is

$$\frac{\Omega \vdash P_x :: (x : A) \quad \Omega_L (x : A) \quad \Omega_R \vdash Q_x :: (z : C)}{\Omega_L \Omega \Omega_R \vdash (x \leftarrow P_x ; Q_x) :: (z : C)} \text{ cut}$$

where we use subscripts to denote which variables are bound in the processes. The other rules are available on the last page of this document, excerpted from the 15-816 Lecture 8 notes from fall 2016. We remark that we have not seen and will not use the rules for the “ $\circ$ ” twist connective.

---

Date: 6 December 2017.

Based in part on Frank Pfenning's 15-816 lecture notes from fall 2016.

## 2. EXAMPLES

We will spend the remainder of the recitation exploring examples involving lists and list segments. First, we recall the definition of  $\text{list}_A$  from Lecture 24:

$$\text{list}_A = \oplus\{\text{cons} : A \bullet \text{list}_A, \text{nil} : \mathbf{1}\}.$$

A list segment is a list prefix, that is to say, a process of the type

$$\text{seg}_A = \text{list}_A / \text{list}_A$$

expecting to receive a tail from the right, and which then provides the concatenation of the prefix with the tail. We will implement several natural list segments.

The first is the *empty* list segment which, given a list on its right, simply produces that list. This leads us to believe that it should be typed  $\cdot \vdash \text{empty} :: (s : \text{seg}_A)$ . Its implementation is then:

$$\begin{aligned} s \leftarrow \text{empty} &= && \% \cdot \vdash s : \text{list}_A / \text{list}_A \\ t \leftarrow \text{recv } s; && \% t : \text{list}_A \vdash s : \text{list}_A \\ s \leftarrow t && & \end{aligned}$$

In the second line, we receive the tail  $t : \text{list}_A$  over the channel  $s$ , and in the third line, we forward from  $t$  to  $s$ . The comments following the “%” sign describe the type of the term we must provide on the remaining lines. For example, on the first line, it denotes that we must provide a body  $\text{empty}(s)$  such that  $\cdot \vdash \text{empty}(s) :: (s : \text{seg}_A)$ , and on the second line that we must provide a term  $T$  such that  $t : \text{list}_A \vdash T :: (s : \text{list}_A)$ .

Another thing we can do is *concatenate* segments. Its type should be:

$$(s_1 : \text{seg}_A)(s_2 : \text{seg}_A) \vdash \text{concat} :: (s : \text{seg}_A).$$

We implement it as follows:

$$\begin{aligned} s \leftarrow \text{concat} \leftarrow s_1 s_2 &= \\ t \leftarrow \text{recv } s; & \% (s_1 : \text{seg}_A)(s_2 : \text{seg}_A)(t : \text{list}_A) \vdash s : \text{list}_A \\ \text{send } s_1 t; & \% (s_1 : \text{seg}_A)(s_2 : \text{list}_A) \vdash s : \text{list}_A \\ \text{send } s_1 s_2; & \% s_1 : \text{list}_A \vdash s : \text{list}_A \\ s \leftarrow s_1 & \end{aligned}$$

We may wish to convert segments to lists by simply providing them with *nil* as a tail. The corresponding type ought to be:  $s : \text{seg}_A \vdash \text{toList} :: (l : \text{list}_A)$ . The implementation is:

$$\begin{aligned} l \leftarrow \text{toList} \leftarrow s &= \\ n \leftarrow \text{nil}; & \% (s : \text{seg}_A)(n : \text{list}_A) \vdash l : \text{list}_A \\ \text{send } s n; & \% s : \text{list}_A \vdash l : \text{list}_A \\ l \leftarrow s & \end{aligned}$$

(Recall from lecture:  $\cdot \vdash \text{nil} :: (l : \text{list}_A)$  given by  $l \leftarrow \text{nil} = l.\text{nil}; \text{close } l$ .)

Conversely, we may wish to convert a list to a list segment. The corresponding type is  $l : \text{list}_A \vdash \text{toSeg} :: (s : \text{seg}_A)$ . The implementation is:

$$\begin{aligned} s \leftarrow \text{toSeq} \leftarrow l &= \\ t \leftarrow \text{recv } l & \% (l : \text{list}_A)(t : \text{list}_A) \vdash s : \text{list}_A \\ \text{concat} & \end{aligned}$$

where we make use of the helper function  $(q : \text{list}_A)(r : \text{list}_A) \vdash concat :: (s : \text{list}_A)$  that concatenates two lists:

```
s ← concat ← q r =  
  case q (nil ⇒ wait q; s ← r  
          | cons ⇒ h ← recv q; s.cons; send s h; concat)
```

**Judgmental Rules**

$$\frac{\Omega \vdash P_x :: (x:A) \quad \Omega_L (x:A) \Omega_R \vdash Q_x :: (z:C)}{\Omega_L \Omega_R \vdash (x \leftarrow P_x ; Q_x) :: (z:C)} \text{ cut} \quad \frac{}{y:A \vdash x \leftarrow y :: (x:A)} \text{id}$$

**Additive Connectives**

$$\frac{\Omega \vdash P :: (x:A_k) \quad (k \in I)}{\Omega \vdash (x.l_k ; P) :: (x : \oplus\{l_i:A_i\}_{i \in I})} \oplus R_k \quad \frac{\Omega_L (x:A_i) \Omega_R \vdash Q_i :: (z:C) \quad (\forall i \in I)}{\Omega_L (x:\oplus\{l_i:A_i\}_{i \in I}) \Omega_R \vdash \text{case } x (l_i \Rightarrow Q_i)_{i \in I} :: (z:C)} \oplus L$$

$$\frac{\Omega \vdash P_i :: (x:A_i) \quad (\forall i \in I)}{\Omega \vdash \text{case } x (l_i \Rightarrow P_i)_{i \in I} :: (x:\&\{l_i:A_i\}_{i \in I}))} \& R \quad \frac{\Omega_L (x:A_k) \Omega_R \vdash P :: (z:C) \quad (k \in I)}{\Omega_L (x : \&\{l_i:A_i\}_{i \in I}) \Omega_R \vdash (x.l_k ; Q) :: (z:C)} \& L_k$$

**Multiplicative Connectives**

$$\frac{}{\cdot \vdash \text{close } x :: (x:\mathbf{1})} \mathbf{1} R \quad \frac{\Omega_L \Omega_R \vdash Q :: (z:C)}{\Omega_L (x:\mathbf{1}) \Omega_R \vdash (\text{wait } x ; Q) :: (z:C)} \mathbf{1} L$$

$$\frac{\Omega (y:B) \vdash P_y :: (x:A)}{\Omega \vdash (y \leftarrow \text{recv } x ; P_y) :: (x:A / B)} / R \quad \frac{\Omega_L (x:A) \Omega_R \vdash Q :: (z:C)}{\Omega_L (x:A / B) (w:B) \Omega_R \vdash (\text{send } x w ; Q) :: (z:C)} / L^*$$

$$\frac{(y:B) \Omega \vdash P_y :: (x:A)}{\Omega \vdash (y \leftarrow \text{recv } x ; P_y) :: (x:B \setminus A)} \setminus R \quad \frac{\Omega_L (x:A) \Omega_R \vdash Q :: (z:C)}{\Omega_L (w:B) (x:B \setminus A) \Omega_R \vdash (\text{send } x w ; Q) :: (z:C)} \setminus L^*$$

$$\frac{\Omega \vdash P :: (x:B)}{(w:A) \Omega \vdash (\text{send } x w ; P) :: (x:A \bullet B)} \bullet R^* \quad \frac{\Omega_L (y:A) (x:B) \Omega_R \vdash Q_y :: (z:C)}{\Omega_L (x:A \bullet B) \Omega_R \vdash (y \leftarrow \text{recv } x ; Q_y) :: (z:C)} \bullet L$$

$$\frac{\Omega \vdash P :: (x:B)}{\Omega (w:A) \vdash (\text{send } x w ; P) :: (x:A \circ B)} \circ R^* \quad \frac{\Omega_L (x:B) (y:A) \Omega_R \vdash Q_y :: (z:C)}{\Omega_L (x:A \bullet B) \Omega_R \vdash (y \leftarrow \text{recv } x ; Q_y) :: (z:C)} \circ L$$

## RECITATION 1

### 1. SYLLABUS

- (1) Lectures are Tuesday and Thursday, GHC 4307, from 1:30-2:50.
- (2) Recitations are not required but strongly encouraged, taking place on Wednesdays.
- (3) Office hours
  - Jon Sterling: Monday 1:30pm-2:30pm, GHC 9225
  - Frank Pfenning: Wednesday, 1:30pm-2:30pm, GHC 7019
  - Ryan Kavanagh, Friday 11:30am-12:30pm, GHC 6207
  - Danny Gratzer, Sat 11:00am-2:00pm, Citadel Teaching Commons, GHC 5th floor
- (4) Course homework should be turned via autolab.
- (5) Course questions should be asked via Piazza (*Please do not email questions to TAs personally*)
- (6) Grades
  - 40% weekly homework, released Tuesday and due Tuesday with no late days.
  - 30% 2 closed-book, in-class midterms, the first being on Thursday September 28th and the second on Thursday, November 9th.
  - 30% the closed-book final.
- (7) Grading cut-offs will be no harsher than 90% for an A, 80% for a B, etc.
- (8) All of this and more may be found on the course website <http://www.cs.cmu.edu/~fp/courses/15317-f17/>

### 2. REVIEW OF THE RULES INTRODUCED IN CLASS

First let us recall the rules that were introduced in lecture.

$$\begin{array}{cccccc}
 \frac{A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} & \frac{A \text{ true}}{A \vee B \text{ true}} & \frac{B \text{ true}}{A \vee B \text{ true}} & \frac{A \wedge B \text{ true}}{A \text{ true}} & \frac{A \wedge B \text{ true}}{B \text{ true}} & \frac{\overline{A \text{ true}}^U \quad \overline{B \text{ true}}^U}{A \supset B \text{ true}}
 \end{array}$$

$$\frac{A \supset B \text{ true} \quad A \text{ true}}{B \text{ true}}$$

### 3. EXAMPLE PROOFS

- (1)  $(A \supset (B \supset C)) \supset (B \supset (A \supset C))$

$$\frac{\overline{A \supset (B \supset C)}^a \quad \overline{A}^c}{\overline{B \supset C}^b}$$

$$\frac{\overline{B \supset C}^b}{\overline{C}^c}$$

$$\frac{\overline{C}^c}{\overline{A \supset C}^b}$$

$$\frac{\overline{A \supset C}^b}{\overline{B \supset (A \supset C)}^a}$$

$$\frac{\overline{B \supset (A \supset C)}^a}{(A \supset B \supset C) \supset (B \supset (A \supset C))^a}$$

(2)  $((A \wedge B) \supset C) \supset (A \supset (B \supset C))$

$$\begin{array}{c}
 \frac{\overline{(A \wedge B) \supset C}^a \quad \overline{\overline{A}^b \quad \overline{B}^c}}{\overline{C}^c} \\
 \hline
 \frac{\overline{B \supset C}^b}{\overline{A \supset (B \supset C)}^a} \\
 \hline
 \frac{\overline{(A \wedge B) \supset C}^a \supset \overline{A \supset (B \supset C)}^a}{(A \wedge B) \supset C \supset (A \supset (B \supset C))}^a
 \end{array}$$

(3)  $((A \wedge B) \wedge C) \supset (A \wedge (B \wedge C))$

$$\begin{array}{c}
 \frac{\overline{(A \wedge B) \wedge C}^a}{\overline{A \wedge B}^a} \quad \frac{\overline{(A \wedge B) \wedge C}^a}{\overline{B}^a} \quad \frac{\overline{(A \wedge B) \wedge C}^a}{\overline{C}^a} \\
 \hline
 \frac{\overline{A}^a \quad \overline{B}^a \quad \overline{C}^a}{B \wedge C} \\
 \hline
 \frac{\overline{(A \wedge (B \wedge C))}^a}{((A \wedge B) \wedge C) \supset (A \wedge (B \wedge C))}^a
 \end{array}$$

(4)  $(A \supset (B \wedge C)) \supset (A \supset B) \wedge (A \supset C)$

$$\begin{array}{c}
 \frac{\overline{A \supset (B \wedge C)}^a \quad \overline{\overline{A}^b}}{\overline{B \wedge C}^b} \quad \frac{\overline{A \supset (B \wedge C)}^a \quad \overline{\overline{A}^c}}{\overline{B \wedge C}^c} \\
 \hline
 \frac{\overline{B}^b \quad \overline{C}^c}{A \supset B \quad A \supset C} \\
 \hline
 \frac{\overline{(A \supset B) \wedge (A \supset C)}^a}{(A \supset (B \wedge C)) \supset (A \supset B) \wedge (A \supset C)}^a
 \end{array}$$

(5) Why is this impossible with the rules we have?  $(A \vee C) \wedge (B \vee C) \supset ((A \wedge B) \vee C)$

## RECITATION 2

### 1. PROOFS ARE PROGRAMS

As discussed previously in lecture, there is a tight correspondence between the structure of a derivation for a constructive proof and a term in some particular programming language. This leads to the slogans “proofs are programs” and “propositions are types”. The correspondence can be fleshed out for the logic we’re studying (intuitionistic propositional logic)<sup>1</sup> by the following table

Propositions	Types
$A \wedge B$	$A * B$
$A \vee B$	$A + B$
$A \supset B$	$A \rightarrow B$
$\top$	$1$ (unit)
$\perp$	$0$ (void)

Based on this we can produce a version of our rules from the previous recitation that annotate each proposition step in the derivation with the program that it constructs. Those rules are

$$\begin{array}{c}
\frac{M : A \text{ true} \quad N : B \text{ true}}{\langle M, N \rangle : A \wedge B \text{ true}} \quad \frac{M : A \text{ true}}{\text{inl}(M) : A \vee B \text{ true}} \quad \frac{M : B \text{ true}}{\text{inr}(M) : A \vee B \text{ true}} \quad \frac{M : A \wedge B \text{ true}}{\text{fst}(M) : A \text{ true}} \\
\\
M : A \vee B \text{ true} \quad \frac{\begin{array}{c} u : A \text{ true} \\ \vdots \\ N : C \text{ true} \end{array}^u \quad \begin{array}{c} v : B \text{ true} \\ \vdots \\ R : C \text{ true} \end{array}^v}{\text{case } M \text{ of inl}(c) \Rightarrow N \mid \text{inl}(c) \Rightarrow R : C \text{ true}}^{u, v} \quad \frac{u : A \text{ true}}{\text{inr } u \Rightarrow M : A \supset B \text{ true}}^u \\
\frac{M : A \wedge B \text{ true}}{\text{snd}(M) : B \text{ true}} \quad \frac{M : A \supset B \text{ true} \quad N : A \text{ true}}{M(N) : B \text{ true}}
\end{array}$$

### 2. TRANSLATION

We now turn to the question of translating proofs to programs and back again. In these notes, we present both for the sake of accessibility.

$$(1) \ (A \supset B \supset C) \supset (A \supset B) \supset A \supset C$$

---

*Date:* September 8, 2017.

<sup>1</sup>Of course, what makes this correspondence so remarkable is that it extends far beyond this one logic. It is quite robust and extends to almost any well-behaved logic.

**Proof:**

$$\begin{array}{c}
 \frac{\overline{A \supset B \supset C}^u \quad \overline{\overline{A}}^w}{B \supset C} \quad \frac{\overline{A \supset B}^v \quad \overline{\overline{A}}^w}{B} \\
 \hline
 \frac{C}{A \supset C} \qquad \qquad \qquad w \\
 \hline
 \frac{A \supset C}{(A \supset B) \supset A \supset C} \qquad v \\
 \hline
 \frac{(A \supset B) \supset A \supset C}{(A \supset B \supset C) \supset (A \supset B) \supset A \supset C} \qquad u
 \end{array}$$

**Program:** fn f => fn g => fn a => f a (g a)

$$(2) (A \supset B \supset C) \supset (B \supset A \supset C)$$

**Proof:**

$$\begin{array}{c}
 \frac{\overline{A \supset (B \supset C)}^a \quad \overline{\overline{A}}^c}{B \supset C} \quad \overline{\overline{B}}^b \\
 \hline
 \frac{C}{A \supset C} \qquad \qquad \qquad c \\
 \hline
 \frac{A \supset C}{B \supset (A \supset C)} \qquad b \\
 \hline
 \frac{(A \supset B \supset C) \supset (B \supset (A \supset C))}{(A \supset B \supset C) \supset (B \supset (A \supset C))} \qquad a
 \end{array}$$

**Program:** fn f => fn a => fn b => f b a

$$(3) A \wedge (B \vee C) \supset (A \wedge B) \vee (A \wedge C)$$

**Proof:**

$$\begin{array}{c}
 \frac{\overline{A \wedge (B \vee C)}^u \quad \overline{\overline{B}}^w}{A \quad \overline{B}^w} \quad \frac{\overline{A \wedge (B \vee C)}^u \quad \overline{\overline{C}}^w}{A \quad \overline{C}^w} \\
 \hline
 \frac{A \wedge (B \vee C)}{B \vee C} \qquad \qquad \qquad \frac{A \wedge (B \vee C)}{A \wedge C} \\
 \hline
 \frac{(A \wedge B) \vee (A \wedge C)}{(A \wedge B) \vee (A \wedge C)} \qquad \qquad \qquad w \\
 \hline
 \frac{(A \wedge B) \vee (A \wedge C)}{A \wedge (B \vee C) \supset (A \wedge B) \vee (A \wedge C)} \qquad \qquad \qquad -u
 \end{array}$$

**Program:** fn x => case snd x of inl b => inl (fst x, b) | inr c => inr (fst x, c)

$$(4) (\top \vee \top) \supset (\top \vee \top) \supset (\top \vee \top)$$

**Proof:**

$$\begin{array}{c}
 \frac{\overline{\top \vee \top}^u \quad \overline{\top \vee \top}^w}{\top \vee \top} \quad \frac{\overline{\top \vee \top}^u \quad \overline{\overline{\top \vee \top}}^w}{\top \vee \top} \\
 \hline
 \frac{\top \vee \top}{(\top \vee \top) \supset (\top \vee \top)} \qquad v \\
 \hline
 \frac{(\top \vee \top) \supset (\top \vee \top)}{(\top \vee \top) \supset (\top \vee \top) \supset (\top \vee \top)} \qquad u
 \end{array}$$

**Program:** fn a => fn b => case a of inl \_ => b | inr \_ => inr ()

$$(5) (\top \vee \top) \supset (\top \vee \top) \supset (\top \vee \top)$$

**Proof:**

$$\begin{array}{c}
 \frac{\overline{\top \vee \top}^u \quad \frac{\overline{\top \vee \top}^v}{\top \vee \top} \quad \overline{\top \vee \top}^w}{\top \vee \top} \\
 \hline
 \frac{\top \vee \top}{(\top \vee \top) \supset (\top \vee \top)} \qquad w \\
 \hline
 \frac{(\top \vee \top) \supset (\top \vee \top)}{(\top \vee \top) \supset (\top \vee \top) \supset (\top \vee \top)} \qquad u
 \end{array}$$

**Program:** fn a => fn b => case a of inl \_ => inl () | inr \_ => b

# Recitation 3: Harmony

Course Staff

Proof-theoretic harmony is a necessary, but not sufficient, condition for the well-behavedness of a logic; harmony ensures that the connectives are *locally* well-behaved, and is closely related to the critical cases of cut and identity elimination which we may discuss later on. Therefore, when designing or extending a logic, checking harmony is a first step.

From the verificationist standpoint, a connective is *harmonious* if its elimination rules are neither too strong nor too weak in relation to its introduction rules. The first condition is called *local soundness* and the second condition is called *local completeness*. The content of the soundness condition is a method to reduce or simplify proofs, and the content of completeness is a method to expand any arbitrary proof into a canonical proof (i.e. one that ends in an introduction rule).

## 1 Conjunction

Local soundness for conjunction is witnessed by the following two reduction rules:

$$\frac{\begin{array}{c} \mathcal{D} \\ A \text{ true} \end{array} \quad \begin{array}{c} \mathcal{E} \\ B \text{ true} \end{array}}{\frac{A \wedge B \text{ true}}{A \text{ true}}} \wedge E_1 \quad \rightarrow_R \quad \boxed{\begin{array}{c} \mathcal{D} \\ A \text{ true} \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{D} \\ A \text{ true} \end{array} \quad \begin{array}{c} \mathcal{E} \\ B \text{ true} \end{array}}{\frac{A \wedge B \text{ true}}{B \text{ true}}} \wedge E_2 \quad \rightarrow_R \quad \boxed{\begin{array}{c} \mathcal{E} \\ B \text{ true} \end{array}}$$

Local completeness is witnessed by the following expansion rule:

$$\boxed{\begin{array}{c} \mathcal{D} \\ A \wedge B \text{ true} \end{array}} \quad \rightarrow_E \quad \frac{\frac{\begin{array}{c} \mathcal{D} \\ A \wedge B \text{ true} \end{array}}{\begin{array}{c} \mathcal{D} \\ A \text{ true} \end{array}} \wedge E_1 \quad \frac{\begin{array}{c} \mathcal{D} \\ A \wedge B \text{ true} \end{array}}{\begin{array}{c} \mathcal{D} \\ B \text{ true} \end{array}} \wedge E_2}{A \wedge B \text{ true}} \wedge I$$

When regarded as generating relations on *programs* rather than proofs, the reduction and expansion rules can be recast into another familiar format:

$$\begin{aligned} \text{fst}(\langle M, N \rangle) &\rightarrow_R M \\ \text{snd}(\langle M, N \rangle) &\rightarrow_R N \\ M &\rightarrow_E \langle \text{fst}(M), \text{snd}(M) \rangle \end{aligned}$$

## 2 Disjunction

Instructions: present local soundness for proofs, and ask the students to come up with the version for programs. Next, elicit from the students both local completeness for programs and for proofs.

$$\begin{array}{c}
 \frac{\mathcal{D} \quad A \text{ true}}{A \vee B \text{ true}} \vee I_1 \quad \frac{\overline{A \text{ true}}^u \quad \mathcal{E} \quad \overline{B \text{ true}}^v}{\overline{C \text{ true}}^v} \quad \frac{\overline{A \text{ true}}^u \quad \mathcal{F} \quad \overline{B \text{ true}}^v}{\overline{C \text{ true}}^v} \\
 \hline
 C \text{ true} \quad \vee E^{u,v} \xrightarrow{R} \frac{\mathcal{D} \quad A \text{ true}}{\overline{C \text{ true}}^u} \quad \frac{\mathcal{D} \quad B \text{ true}}{\overline{C \text{ true}}^v}
 \end{array}$$

$\frac{\mathcal{D} \quad B \text{ true}}{A \vee B \text{ true}} \vee I_2 \quad \frac{\overline{A \text{ true}}^u \quad \mathcal{E} \quad \overline{B \text{ true}}^v}{\overline{C \text{ true}}^v}$        $\frac{\mathcal{D} \quad B \text{ true}}{\overline{C \text{ true}}^v} \xrightarrow{R} \frac{\mathcal{D} \quad B \text{ true}}{\overline{C \text{ true}}^v}$

$\text{case inl}(\boxed{M}) \text{ of inl}(u) \Rightarrow \boxed{L} \mid \text{inr}(v) \Rightarrow R \xrightarrow{R} [\boxed{M} / u]L$   
 $\text{case inr}(\boxed{M}) \text{ of inl}(u) \Rightarrow L \mid \text{inr}(v) \Rightarrow R \xrightarrow{R} [\boxed{M} / v]R$

$$\frac{\mathcal{D} \quad A \vee B \text{ true}}{A \vee B \text{ true}} \xrightarrow{E} \frac{\frac{\mathcal{D} \quad A \text{ true}}{A \vee B \text{ true}} \quad \frac{\mathcal{D} \quad B \text{ true}}{A \vee B \text{ true}}}{A \vee B \text{ true}} \vee I_1 \quad \frac{\mathcal{D} \quad B \text{ true}}{A \vee B \text{ true}} \vee I_2$$

$\boxed{M} \xrightarrow{E} \text{case } \boxed{M} \text{ of inl}(u) \Rightarrow \text{inl}(u) \mid \text{inr}(v) \Rightarrow \text{inr}(v)$

## 3 Implication

Elicit both local soundness and local completeness from students in both proof and program notation.

$$\frac{\overline{A \text{ true}}^u \quad \mathcal{D} \quad B \text{ true}}{A \supset B \text{ true}} \supset I^u \quad \frac{\mathcal{E} \quad A \text{ true}}{A \supset B \text{ true}} \supset E \xrightarrow{R} \frac{\mathcal{E} \quad A \text{ true}}{B \text{ true}}$$

$(\text{fn } u \Rightarrow \boxed{M})(\boxed{N}) \xrightarrow{R} [\boxed{N} / u]M$

$$\frac{\mathcal{D} \quad A \supset B \text{ true}}{A \supset B \text{ true}} \xrightarrow{E} \frac{\overline{A \text{ true}}^u \quad \mathcal{D} \quad B \text{ true}}{A \supset B \text{ true}} \supset I^u$$

$\boxed{M} \xrightarrow{E} \text{fn } u \Rightarrow \boxed{M}(u)$

## 4 Experiment: Alternative Implication

What if we replaced the  $\supset E$  rule with the following elimination rule:

$$\frac{\overline{B \text{ true}}^u}{\vdots} \\ \frac{A \supset B \text{ true} \quad A \text{ true} \quad C \text{ true}}{C \text{ true}} \supset E^u$$

The program/proof term assignment is as follows:

$$\frac{\overline{u : B}^u}{\vdots} \\ \frac{L : A \supset B \quad M : A \quad N : C}{\text{let } u = L(M) \text{ in } N : C} \supset E^u$$

Can we show local soundness and completeness for this version of the implication connective?

$$\frac{\overline{A \text{ true}}^v \quad \frac{\overline{D}^v \quad \overline{B \text{ true}}}{\vdots} \supset I^v \quad \frac{\mathcal{E} \quad A \text{ true}}{B \text{ true}} \quad \frac{\overline{B \text{ true}}^u \quad \frac{\mathcal{F} \quad C \text{ true}}{\vdots} \supset E^u}{C \text{ true}}}{A \supset B \text{ true}} \longrightarrow_R \frac{\mathcal{E} \quad A \text{ true} \quad \frac{\overline{D}^v \quad \overline{B \text{ true}}^u}{\vdots} \supset E^u \quad \frac{\mathcal{F} \quad C \text{ true}}{\vdots}}{C \text{ true}}$$

let  $u = \text{fn } v \Rightarrow L(M) \text{ in } N \longrightarrow_R [[M / v]L / u]N$

$$\frac{\overline{A \supset B \text{ true}}^{\mathcal{D}} \quad \frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^v}{\vdots} \supset E^v}{A \supset B \text{ true}} \longrightarrow_E \quad \frac{\overline{B \text{ true}}}{\overline{A \supset B \text{ true}}} \supset I^u$$

$M \longrightarrow_E \text{fn } u \Rightarrow \text{let } v = M(u) \text{ in } v$

# Recitation 4: Verifications and Quantifiers

Jon Sterling

“Proof search” is not a mere matter of practice: it is *praxis*. The dialectic of proof search is to discover ways to pare down the state space of a logic, and then synthesize this into a new logic which is exactly as expressive as the old one. This new restricted logic not only has better search complexity, but also exposes critical semantic content which tend to have been obscured in the original logic.

Perhaps the most famous example of this process is Andreoli’s *focalization*; in recent lectures, we have begun to study a simpler instance of this process, namely the decomposition of truth into *verification* and *use*. The passage to verifications constitutes a collation of upward and downward deductions respectively.

## 1 Verifications and Uses

“Verifications” are proofs that proceed upwards from conclusions to premises; this is also known as *backward inference* or *refinement-style proof*. On the other hand, “uses” are proofs that proceed from premise to conclusion, also known as *forward inference*. The judgment  $\boxed{A \uparrow}$  stands for verifications of  $A$ , and the judgment  $\boxed{A \downarrow}$  stands for uses of  $A$ .

The rules for verifications and uses of the conjunction connective are as follows:

$$\frac{\boxed{A \uparrow} \quad \boxed{B \uparrow}}{\boxed{A \wedge B \uparrow}} \wedge I \qquad \frac{\boxed{A \wedge B \downarrow}}{\boxed{A \downarrow}} \wedge E_1 \qquad \frac{\boxed{A \wedge B \downarrow}}{\boxed{B \downarrow}} \wedge E_2$$

On this basis, you may think that verifications correspond to introduction forms and uses correspond to elimination forms. This is not correct, as can be seen from the case of disjunction:

$$\frac{\boxed{A \uparrow} \quad \boxed{B \uparrow}}{\boxed{A \vee B \uparrow}} \vee I_1 \qquad \frac{\boxed{A \vee B \uparrow}}{\boxed{A \downarrow} \quad \boxed{B \downarrow}} \vee I_2 \qquad \frac{\boxed{A \vee B \downarrow} \quad \boxed{C \uparrow} \quad \boxed{C \uparrow}}{\boxed{C \uparrow} \quad \boxed{C \uparrow}} \vee E^{u,v}$$

**Question.** Will the elimination rule for implication result have a verification or a use in its conclusion?

$$\frac{\overline{A \downarrow} \quad u}{\begin{array}{c} \vdots \\ B \uparrow \end{array}} \supset I^u \qquad \frac{A \supset B \downarrow \quad A \uparrow}{B \downarrow} \supset E$$

**Remark (Bonus).** One dimension along which connectives vary is polarity: some connectives are positive, and some are negative. We cannot yet make this distinction precise, but some students have already begun to observe it. Later on, we may see that negative connectives have elimination forms as uses, but positive connectives have elimination forms as verifications.

The calculus of verifications and uses has one extra rule which was not visible in the original logic:

$$\frac{A \downarrow}{A \uparrow} \uparrow$$

**Question.** Would it be reasonable to add the inverse of the above rule, which concludes  $A \downarrow$  from  $A \uparrow$ ? What would be the consequences of this?

We have also begun to study quantifiers (universal and existential). The rules for these are as follows:

$$\frac{\overline{c : A}}{\begin{array}{c} \vdots \\ A(c) \uparrow \end{array}} \forall I^c \qquad \frac{\forall x : \tau. A(x) \downarrow \quad t : \tau}{A(t) \downarrow} \forall E$$
  

$$\frac{t : \tau \quad A(t) \uparrow}{\exists x : \tau. A(x) \uparrow} \exists I \qquad \frac{\overline{c : \tau} \quad \overline{A(c) \downarrow} \quad u}{\begin{array}{c} \vdots \\ C \uparrow \end{array}} \exists E^{c,u}$$

## 2 Examples with quantifiers

Consider a predicate  $A(x)$  which depends on  $x : \tau$  and a proposition  $B$ .

### 2.1 Existential Adjointness

Prove the following equivalence in the logic of verifications and uses:

$$(\forall x : \tau. (A(x) \supset B)) \equiv ((\exists x : \tau. A(x)) \supset B) \uparrow$$

**Remark.** For the advanced, the above exercise is essentially the fact that the existential quantifier is “left adjoint” to weakening.

*Proof.* First the proof from left to right:

$$\begin{array}{c}
 \frac{\overline{\forall x : \tau. (A(x) \supset B) \downarrow}^u \quad \overline{c : \tau}^w}{\overline{A(c) \supset B \downarrow}^{\forall E}} \\
 \frac{\overline{\exists x : \tau. A(x) \downarrow}^v \quad \frac{\overline{B \downarrow}^{\uparrow} \quad \overline{B \uparrow}^{\uparrow}}{\exists E^{c,w}}}{\overline{B \uparrow}^{\supset I^v}} \\
 \frac{\overline{(\exists x : \tau. A(x)) \supset B \uparrow}^{\supset I^u}}{(\forall x : \tau. (A(x) \supset B)) \supset ((\exists x : \tau. A(x)) \supset B) \uparrow} \quad (\Rightarrow)
 \end{array}$$

Next, the proof from right to left:

$$\begin{array}{c}
 \frac{\overline{A(c) \downarrow}^v \quad \overline{A(c) \uparrow}^{\uparrow}}{\overline{c : \tau}^u \quad \overline{\exists x : \tau. A(x) \uparrow}^{\exists I}} \\
 \frac{\overline{B \downarrow}^{\uparrow} \quad \overline{B \uparrow}^{\supset I^v}}{\overline{A(c) \supset B \uparrow}^{\forall I^c}} \\
 \frac{\overline{\forall x : \tau. (A(x) \supset B) \uparrow}^{\supset I^u}}{((\exists x : \tau. A(x)) \supset B) \supset (\forall x : \tau. (A(x) \supset B)) \uparrow} \quad (\Leftarrow)
 \end{array}$$

□

## 2.2 Universal Adjunctness

Prove the following equivalence in the logic of verifications and uses:

$$(\forall x : \tau. (B \supset A(x))) \equiv (B \supset \forall x : \tau. A(x)) \uparrow$$

**Remark.** Likewise, this is the fact that the universal quantifier is “right adjoint” to weakening.

*Proof.* First the proof from left to right:

$$\begin{array}{c}
 \frac{\overline{\forall x : \tau. (B \supset A(x)) \downarrow}^u \quad \overline{c : \tau}^w \quad \overline{B \downarrow}^{\uparrow} \quad \overline{B \uparrow}^v}{\overline{B \supset A(c) \downarrow}^{\forall E}} \\
 \frac{\overline{A(c) \downarrow}^{\uparrow} \quad \overline{A(c) \uparrow}^{\uparrow}}{\overline{\forall x : \tau. A(x) \uparrow}^{\forall I^c}} \\
 \frac{\overline{B \supset \forall x : \tau. A(x) \uparrow}^{\supset I^v}}{(\forall x : \tau. (B \supset A(x))) \supset (B \supset \forall x : \tau. A(x)) \uparrow} \quad (\Rightarrow)
 \end{array}$$

Next, the proof from right to left:

$$\frac{\frac{\frac{\frac{B \supset \forall x : \tau. A(x) \downarrow}{\forall x : \tau. A(x) \downarrow} u \quad \frac{\overline{B \downarrow} \quad v}{\overline{B \uparrow} \quad \uparrow} \supset E}{\forall x : \tau. A(x) \downarrow} \quad \overline{c : \tau}}{\frac{\frac{\overline{A(c) \downarrow} \quad \uparrow}{\overline{A(c) \uparrow} \quad \uparrow} \supset I^v}{\frac{\frac{B \supset A(c) \uparrow}{\forall x : \tau. (B \supset A(x)) \uparrow} \quad \forall I^c}{(B \supset \forall x : \tau. A(x)) \supset (\forall x : \tau. (B \supset A(x))) \uparrow} \supset I^u}}}{(B \supset \forall x : \tau. A(x)) \supset (\forall x : \tau. (B \supset A(x))) \uparrow} \supset I^u} \quad ( \Leftarrow )$$

□

### 2.3 Swapping Quantifiers

If there is time, try proving that an existential quantification can be moved underneath a universal quantification. Fixing a predicate in two variables  $A(x, y)$  for  $x : \sigma, y : \tau$ :

$$\frac{\frac{\frac{\frac{\forall x : \sigma. A(x, d) \downarrow \quad v}{\overline{c : \sigma}} \quad \forall E}{\frac{\overline{A(c, d) \downarrow} \quad \uparrow}{\overline{A(c, d) \uparrow} \quad \uparrow} \quad \exists I}{\frac{\frac{\overline{d : \tau}}{\exists y : \tau. A(c, y) \uparrow} \quad \forall I^c}{\frac{\frac{\exists y : \tau. A(c, y) \uparrow}{\forall x : \sigma. \exists y : \tau. A(x, y) \uparrow} \quad \exists E^{d,v}}{( \exists y : \tau. \forall x : \sigma. A(x, y) ) \supset ( \forall x : \sigma. \exists y : \tau. A(x, y) ) \uparrow} \supset I^u}}}{( \exists y : \tau. \forall x : \sigma. A(x, y) ) \supset ( \forall x : \sigma. \exists y : \tau. A(x, y) ) \uparrow} \supset I^u} \quad ( \Leftarrow )$$

## RECITATION 5: INDUCTION, PRIMITIVE RECURSION, & MIDTERM REVIEW

RYAN KAVANAGH

### 1. INDUCTION & PRIMITIVE RECURSION

**1.1. A brief recapitulation of Lecture 8.** In Lecture 8 (and yesterday's review!), we saw two different elimination rules for natural numbers. The first, which captures induction, is a judgmental form of the principle of induction:

$$\frac{\begin{array}{c} \overline{x : \text{nat}} \quad \overline{C(x) \text{ true}} \quad u \\ \vdots \\ n : \text{nat} \quad C(0) \text{ true} \quad C(sx) \text{ true} \end{array}}{C(n) \text{ true}} \text{ natE}^{x,u}$$

The other was the *rule of primitive recursion*, which introduces a new term constructor  $R$  for each type  $\tau$ :

$$\frac{\begin{array}{c} \overline{x : \text{nat}} \quad \overline{r : \tau} \\ \vdots \\ n : \text{nat} \quad t_0 : \tau \quad t_s : \tau \end{array}}{R(n, t_0, x.r.t_s) : \tau} \text{ natE}^{x,r}$$

Its behaviour is captured by the following reduction rules:

$$\begin{aligned} R(0, t_0, x.r.t_s) &\Longrightarrow_R t_0, \\ R(s n', t_0, x.r.t_s) &\Longrightarrow_R [R(n', t_0, x.r.t_s)/r][n'/x] t_s. \end{aligned}$$

These rules  $R$  indicate that  $R$  describes a recursive function “ $R(n)$ ” on the first parameter, with value  $t_0$  when  $n = 0$ , and value  $[R(n')/r][n'/x]t_s$  when  $n = s n'$ . This motivates the more readable *schema of primitive recursion*, where we define the function (call it “ $f$ ” to avoid confusion)  $f$  by cases:

$$\begin{aligned} f(0) &= t_0, \\ f(sx) &= t_s(x, f(x)). \end{aligned}$$

We can recover the recursor version of the definition as follows:

$$f = (\text{fn } n \Rightarrow R(n, t_0, x.r.t_s(x, r))).$$

### 1.2. Working with these ideas.

**Exercise 1.** The judgmental form of the principle of induction can be used to show the following more traditional formulation that uses universal quantification:

$$\forall n : \text{nat}. C(0) \supset (\forall x : \text{nat}. C(x) \supset C(sx)) \supset C(n) \text{ true}.$$

What is the corresponding proof term?

*Solution.*

$$\begin{array}{c}
 \frac{\forall x : \text{nat}. C(x) \supset C(sx) \quad x : \text{nat}}{C(x) \supset C(sx)} \forall E \quad \frac{}{C(sx)} \supset E \\
 \frac{n : \text{nat} \quad \frac{}{C(o)} u}{C(n)} \text{ natE}^{x,w} \\
 \frac{\frac{\forall x : \text{nat}. C(x) \supset C(sx) \supset C(n)}{C(o) \supset (\forall x : \text{nat}. C(x) \supset C(sx)) \supset C(n)} \supset I^v}{\forall n : \text{nat}. C(o) \supset (\forall x : \text{nat}. C(x) \supset C(sx)) \supset C(n)} \supset I^n
 \end{array}$$

The corresponding proof term is  $\text{fn } n \Rightarrow \text{fn } u \Rightarrow \text{fn } v \Rightarrow R(n, u, x.w.(vx)w)$ .  $\square$

The *total predecessor function* is the primitive recursive function given by the primitive recursion schema

$$\begin{aligned}
 \text{pred}(o) &= o \\
 \text{pred}(sx) &= x,
 \end{aligned}$$

or equivalently,

$$\text{pred}(n) = R(n, o, x.r.x).$$

**Exercise 2.** Informally prove  $\forall x : \text{nat}. x = o \vee s(\text{pred } x) = x$ . Extract the corresponding function, assuming we already have a proof term  $p : \forall x. \text{pred } (sx) = x$ .

*Proof.* By induction on  $x$ .

- Case  $x = o$ . Need to show:  $o = o \vee s(\text{pred } o) = o$ . We are done by  $=I_{oo}$  and  $\vee I_1$ .
- Case  $x = sx'$ . Assume:  $x' = o \vee s(\text{pred } x') = sx'$ . By definition of  $\text{pred}$ , we have  $\text{pred } sx' = x'$ . By  $=I_{ss}$ ,  $s(\text{pred } sx') = sx'$ . By  $\vee I_2$ ,  $sx' = o \vee s(\text{pred } sx') = sx'$ . This is what we wanted to show.

The corresponding proof term is:  $\text{fn } x \Rightarrow R(x, \text{inl} (=I_{oo}), x'.r.\text{inr} (=I_{ss}(px')))$ .  $\square$

We define *proper subtraction* for the natural numbers as follows to be the function

$$a \div b = \begin{cases} a - b & a \geq b \\ o & a < b. \end{cases}$$

**Exercise 3.** Give a primitive recursive definition for  $\text{pminus}$ .

*Solution.* We are trying to define a function

$$\text{pminus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$$

by primitive recursion such that “ $\text{pminus } a b$ ” encodes  $a \div b$ . We make use of the following (informal) observation:

$$\forall a : \text{nat}. \forall b : \text{nat}. s b \div s c = b \div c.$$

When solving these examples, it is often useful to informally write out what the function should look like, before trying to find a primitive recursive definition:

$$\begin{aligned}
 (\text{pminus } o)(o) &= o, \\
 (\text{pminus } sx)(o) &= sx, \\
 (\text{pminus } o)(sy) &= o, \\
 (\text{pminus } sx)(sy) &= (\text{pminus } x)(y).
 \end{aligned}$$

We can define pminus by primitive recursion on the second argument using the primitive recursion schema as follows:

$$\begin{aligned} (\text{pminus } a)(\text{o}) &= a, \\ (\text{pminus } a)(\text{s } y) &= (\text{pminus } (\text{pred } a))(y), \end{aligned}$$

or using the recursor:

$$\text{pminus } a \ b = R(b, a, x.r. \text{pminus } (\text{pred } a) \ x).$$

Alternatively, we can define pminus by primitive recursion on the first argument:

$$\begin{aligned} \text{pminus } \text{o} &= \text{fn } b \Rightarrow \text{o}, \\ \text{pminus } \text{s } x &= \text{fn } b \Rightarrow R(b, \text{s } x, y.r. \text{pminus } x \ y), \end{aligned}$$

or using the recursor:

$$\text{pminus } a = \text{fn } b \Rightarrow R(a, \text{o}, x.r.R(b, \text{s } x, y.t. \text{pminus } x \ y)).$$

We can quickly check that the recursor definition matches the above informal description. To help you figure out what's going on, we colour-code  $a$  and  $b$ . The results of substitutions are determined by blue and Apricot colour-coding:

$$\begin{aligned} R(\text{o}, \text{o}, x.r.R(\text{b}, \text{s } x, y.t. \text{pminus } x \ y)) &\xrightarrow{R} \text{o}, \\ R(\text{s } x, \text{o}, x.r.R(\text{o}, \text{s } x, y.t. \text{pminus } x \ y)) &\xrightarrow{R} R(\text{o}, \text{s } x, y.t. \text{pminus } x \ y) \\ &\quad \xrightarrow{R} \text{s } x, \\ R(\text{s } x, \text{o}, x.r.R(\text{s } y, \text{s } x, y.t. \text{pminus } x \ y)) &\xrightarrow{R} R(\text{s } y, \text{s } x, y.t. \text{pminus } x \ y) \\ &\quad \xrightarrow{R} \text{pminus } x \ y. \end{aligned} \quad \square$$

## 2. MIDTERM REVIEW

We spend the remainder of the recitation answering the questions that were submitted via Piazza for the review. We will address them in quasi-logical order.

**2.1. Scoping.** Colour-code boxes indicate the scope of each assumption or parametric judgment. Nested coloured boxes indicate that each of the corresponding judgments is in scope.

$$\begin{array}{c} \frac{\overline{B \text{ true}}^u}{A \text{ true}} \vdash^u \\ \hline A \supset B \text{ true} \quad \vdash^u \end{array} \quad \begin{array}{c} \frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^v}{C \text{ true}} \quad \vdash^{u,v} \\ \hline C \text{ true} \quad \vdash^{u,v} \end{array}$$

$$\begin{array}{c} \frac{\overline{x : \tau}}{A(x) \text{ true}} \quad \vdash^x \\ \hline \forall x : \tau. A(x) \text{ true} \quad \forall \vdash^x \end{array} \quad \begin{array}{c} \frac{\exists x : \tau. A(x) \text{ true}}{C \text{ true}} \quad \exists \vdash^x \\ \hline \frac{\overline{a : \tau} \quad \overline{A(a) \text{ true}}^u}{C \text{ true}} \quad \exists E^{\overline{a}, u} \quad \vdash^x \end{array}$$

We would like to emphasise that you are *not* required to use an assumption or parametric judgment. Indeed, when such judgments are in scope, you are free to use them as many times as you wish, including *zero times*. To underscore this point, consider the following exercise:

**Exercise 4.** Prove  $A \supset \top$  true.

*Solution.* Observe that the assumption

$$\overline{A \text{ true}}^u$$

is *not used anywhere* in the following proof:

$$\frac{\overline{\top}^{\top\mid}}{A \supset \top \text{ true}} \supset I^u. \quad \square$$

**2.2. Quantifiers.** By popular demand, we prove properties similar to those you proved on homework 3.

**Exercise 5.** Prove and give the corresponding proof term for  $(\forall x : \tau. A(x)) \supset \neg \exists x. \neg A(x) \text{ true}$ .

*Solution.*

$$\frac{\overline{\exists x : \tau. \neg A(x) \text{ true}}^v \quad \overline{\neg A(a) \text{ true}}^w \quad \frac{\overline{\forall x : \tau. A(x) \text{ true}}^u \quad \overline{a : \tau}}{A(a) \text{ true}} \forall E}{\frac{\perp \text{ true}}{\exists E^{a,w}}} \supset E$$

$$\frac{\perp \text{ true}}{\neg \exists x. \neg A(x) \text{ true}} \neg I^v$$

$$\frac{}{(\forall x : \tau. A(x)) \supset \neg \exists x. \neg A(x) \text{ true}} \supset I^u$$

The corresponding proof term is:  $\text{fn } u \Rightarrow \text{fn } v \Rightarrow \text{let } (a, w) = v \text{ in } w(ua)$ .  $\square$

**Exercise 6.** Give the proof and proof term for

$$(\forall x : \tau. P(x) \supset Q(x)) \supset (\exists x : \tau. \neg Q(x)) \supset \neg \forall x : \tau. P(x) \text{ true}.$$

*Solution.*

$$\frac{\overline{\forall x : \tau. P(x) \supset Q(x)}^u \quad \overline{a : \tau} \quad \forall E \quad \overline{\forall x : \tau. P(x)}^r \quad \overline{a : \tau} \quad \forall E}{\frac{P(a) \supset Q(a)}{\frac{\neg Q(a)}{Q(a)}} \supset E}$$

$$\frac{\overline{\exists x : \tau. \neg Q(x)}^v \quad \frac{\perp}{\neg \forall x : \tau. P(x)} \supset I^r \quad \exists E^{a,w}}{\frac{\neg \forall x : \tau. P(x)}{(\exists x : \tau. \neg Q(x)) \supset \neg \forall x : \tau. P(x)} \supset I^v}$$

$$\frac{}{(\forall x : \tau. P(x) \supset Q(x)) \supset (\exists x : \tau. \neg Q(x)) \supset \neg \forall x : \tau. P(x)} \supset I^u$$

The corresponding proof term is:  $\text{fn } u \Rightarrow \text{fn } v \Rightarrow \text{let } (a, w) = v \text{ in fn } r \Rightarrow w((ua)(ra))$ .  $\square$

**2.3. Harmony.** Consider the “?” connective, defined by its elimination rule:

$$\frac{?(A, B, C) \text{ true} \quad A \text{ true} \quad B \text{ true}}{C \text{ true}} ?E.$$

**Exercise 7.** Give an introduction rule for  $?(A, B, C)$  and show it to be locally sound and complete.

*Solution.*

$$\frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^v \quad \vdots \quad \overline{C \text{ true}}}{?(A, B, C) \text{ true}} ?I^{u,v}$$

Locally sound:

$$\frac{\overline{A \text{ true}} \ u \ \overline{B \text{ true}} \ v}{\begin{array}{c} \mathcal{F} \\ \overline{C \text{ true}} \ ?!^{u,v} \end{array}} \quad \frac{\mathcal{D} \quad \mathcal{E}}{A \text{ true} \quad B \text{ true}} \quad ?\mathbb{E} \xrightarrow{R} \frac{\overline{A \text{ true}} \ u \ \overline{B \text{ true}} \ v}{\begin{array}{c} \mathcal{D} \\ \mathcal{F} \\ C \text{ true} \end{array}}$$

Locally complete:

$$\frac{\mathcal{D}}{?(A, B, C) \text{ true}} \quad \frac{\overline{A \text{ true}} \ u \ \overline{B \text{ true}} \ v}{\begin{array}{c} \mathcal{D} \\ \overline{C \text{ true}} \ ?!^{u,v} \end{array}} \quad ?\mathbb{E} \quad \square$$

## RECITATION 6: SEQUENT CALCULUS

RYAN KAVANAGH

Yesterday, we presented the sequent calculus as a formalism that will be useful in proof search. We saw that there is a tight correspondence between its rules, and the rules for giving verifications for natural deduction. Today, we will review the rules for the sequent calculus and work through a few example proofs.

### 1. THE RULES

Recall that left rules correspond to “upside down elimination rules” and that right rules correspond to introduction rules.

$$\frac{\Gamma, A \wedge B, A \Rightarrow C}{\Gamma, A \wedge B \Rightarrow C} \wedge L_1 \quad \frac{\Gamma, A \wedge B, B \Rightarrow C}{\Gamma, A \wedge B \Rightarrow C} \wedge L_2 \quad \frac{\Gamma \Rightarrow A \quad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B} \wedge R$$

$$\frac{\Gamma, A \vee B, A \Rightarrow C \quad \Gamma, A \vee B, B \Rightarrow C}{\Gamma, A \vee B \Rightarrow C} \vee L \quad \frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B} \vee R_1 \quad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \vee B} \vee R_2$$

$$\text{No } \top L. \quad \frac{}{\Gamma \Rightarrow \top} \top R \quad \frac{}{\Gamma, \perp \Rightarrow C} \perp L \quad \text{No } \perp R.$$

$$\frac{\Gamma, A \supset B \Rightarrow A \quad \Gamma, A \supset B, B \Rightarrow C}{\Gamma, A \supset B \Rightarrow C} \supset L \quad \frac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \supset B} \supset R$$

$$\frac{}{\Gamma, P \Rightarrow P} \text{init}$$

### 2. SOME EXAMPLE PROOFS

We will spend the remainder of the recitation working through some example proofs.

**Exercise 1.**  $\cdot \Rightarrow A \supset A$

*Proof.*

$$\frac{\cdot \Rightarrow A \Rightarrow A \text{ init}}{\cdot \Rightarrow A \supset A} \supset R$$

$\square$

**Exercise 2.**  $\cdot \Rightarrow A \wedge B \supset B \wedge A$

---

*Date:* 4 October 2017.

Most example problems are taken from 15-317 (Fall 2015) Recitation 5, whose notes were prepared by Evan Cavallo, Oliver Duids, and Giselle Reis. Responsibility for any errors herein lies with the present author.

*Proof.*

$$\frac{\overline{A \wedge B, B \Rightarrow B} \text{ init} \quad \overline{A \wedge B, A \Rightarrow A} \text{ init}}{\overline{A \wedge B \Rightarrow B} \wedge_{L_2} \overline{A \wedge B \Rightarrow A} \wedge_{R_1}} \\ \frac{}{\overline{A \wedge B \Rightarrow B \wedge A} \supset R} \\ \cdot \Rightarrow A \wedge B \supset B \wedge A \quad \square$$

**Exercise 3.**  $\cdot \Rightarrow (A \supset (B \wedge C)) \supset (A \supset B)$

*Proof.*

$$\frac{\overline{(A \supset (B \wedge C)), A \Rightarrow A} \text{ init} \quad \overline{(A \supset (B \wedge C)), A, B \wedge C, B \Rightarrow B} \text{ init}}{\overline{(A \supset (B \wedge C)), A, B \wedge C \Rightarrow B} \supset_{L_1}} \\ \frac{}{(A \supset (B \wedge C)), A \Rightarrow B \supset R} \\ \frac{}{(A \supset (B \wedge C)) \Rightarrow (A \supset B) \supset R} \\ \cdot \Rightarrow (A \supset (B \wedge C)) \supset (A \supset B) \quad \square$$

Recall the following example from Recitation 1.

**Exercise 4.**  $\cdot \Rightarrow (A \supset B \supset C) \supset B \supset A \supset C$

*Proof.*

$$\frac{\overline{A \supset B \supset C, B, A \Rightarrow A} \text{ init} \quad \overline{\overline{A \supset B \supset C, B, A, B \supset C \Rightarrow B} \text{ init} \quad \overline{A \supset B \supset C, B, A, B \supset C \Rightarrow C} \text{ init}}{\overline{A \supset B \supset C, B, A, B \supset C \Rightarrow C} \supset_{L_1}} \\ \frac{}{\overline{A \supset B \supset C, B, A \Rightarrow C} \supset R} \\ \frac{\overline{A \supset B \supset C, B \Rightarrow A \supset C} \supset R}{\overline{A \supset B \supset C \Rightarrow B \supset A \supset C} \supset R} \\ \cdot \Rightarrow (A \supset B \supset C) \supset B \supset A \supset C \quad \square$$

**Exercise 5.**  $\cdot \Rightarrow (A \supset B) \supset ((A \wedge C) \supset (B \wedge C))$

*Proof.*

$$\frac{\overline{(A \supset B), (A \wedge C), A \Rightarrow A} \text{ init}}{\overline{(A \supset B), (A \wedge C) \Rightarrow A} \wedge_{L_1}} \quad \frac{\overline{(A \supset B), (A \wedge C), B \Rightarrow B} \text{ init}}{\overline{(A \supset B), (A \wedge C) \Rightarrow B} \supset L} \quad \frac{\overline{(A \supset B), (A \wedge C), C \Rightarrow C} \text{ init}}{\overline{(A \supset B), (A \wedge C) \Rightarrow C} \wedge_{L_2}} \\ \frac{}{(A \supset B), (A \wedge C) \Rightarrow B} \\ \frac{}{(A \supset B), (A \wedge C) \Rightarrow B \wedge C} \\ \frac{}{(A \supset B) \Rightarrow ((A \wedge C) \supset (B \wedge C)) \supset R} \\ \cdot \Rightarrow (A \supset B) \supset ((A \wedge C) \supset (B \wedge C)) \quad \square$$

**Exercise 6.** Poll class for problems or questions if there is time left.

# Implementing Inference Rules in **Standard ML**

Jon Sterling

October 11, 2017

Operationalizing inference rules in a computer program for proof checking or proof search is an important skill. In this tutorial, we will explain and demonstrate some basic techniques for implementing forward and backward inference in the **LCF** style [2, 4, 3].

When implementing a logical system in a programming language, it is important to understand and minimize the size of the portion of this system which must be “trusted”, i.e. on which the correctness of the implementation depends. This is usually achieved by designing a *trusted kernel* with an abstract type together with some operations for constructing elements of that type; then, the only way to produce an object of this type is by calling the provided functions.

An **LCF** kernel consists in such an abstract type **proof**, together with functions which construct proofs according to the rules of inference of the logic. Then, you can use any programming techniques you want (even unsafe ones) to produce proofs, and any such proof which is actually produced is guaranteed to be correct relative to the correctness of the kernel.

## 1 Representing Syntax

The syntax of propositions and sequents in represented in **SML** using *datatypes*. For instance, we can represent the syntax of propositional logic as follows:

```
datatype prop =
  TRUE          (* ⊤ *)
| FALSE         (* ⊥ *)
| ATOM of string (* A *)
| CONJ of prop * prop (* A ∧ B *)
| DISJ of prop * prop (* A ∨ B *)
| IMPL of prop * prop (* A ⊃ B *)
```

It is often convenient to use infix notation in **SML** for the connectives; but note that you need to declare the fixity of these operators.

```
datatype prop =
  TRUE          (* ⊤ *)
| FALSE         (* ⊥ *)
```

```

| ` of string      (* A *)
| /\ of prop * prop  (* A ∧ B *)
| ∨ of prop * prop  (* A ∨ B *)
| ~> of prop * prop  (* A ⊃ B *)

infixr 3 ~>
infixr 4 /\ ∨

```

Contexts are represented as lists of propositions, and we represent sequents as a context together with a proposition:

```

type context = prop list
datatype sequent = ===> of context * prop (* Γ ==> A *)
infixr 0 ===>

```

**Example 1.1** (Structural recursion). Using pattern matching in **SML**, we can write a function that processes the syntax of propositions. Here is such a function which counts how deep a proposition is:

```

val rec depth =
  fn TRUE => 0
  | FALSE => 0
  | `_ => 0
  | a /\ b => Int.max (depth a, depth b) + 1
  | a ∨ b => Int.max (depth a, depth b) + 1
  | a ~> b => Int.max (depth a, depth b) + 1

```

Note that the above is only a more compact notation for the following equivalent **SML** program:

```

fun depth TRUE = 0
| depth FALSE = 0
| depth (`_) = 0
| depth (a /\ b) = Int.max (depth a, depth b) + 1
(* ... *)

```

**Exercise 1.1** (Pretty printing). Write a function to convert propositions into strings, adding parentheses in exactly the necessary and sufficient places according to the precedence of the grammar of propositions. Your solution need not account for associativity.

## 2 Forward Inference Kernel

Usually the trusted kernel of a **LCF**-based proof assistant consists in an implementation of *forward inference*, which is inference from premises to conclusion. We begin with the *signature* for a **LCF** kernel of the intuitionistic sequent calculus; in **SML**, signatures serve as type specifications for entire *structures* (modules).

```

signature KERNEL =
sig
  type proof

  (* What sequent is this a proof of? *)
  val infer : proof -> sequent

```

We represent hypotheses as indices into the context.

```
type hyp = int
```

Next, we give signatures for the rules of intuitionistic sequent calculus, as functions on the type proof; these functions may take additional parameters which are necessary in order to ensure that an actual sequent calculus derivation is uniquely determined by a value of type proof.

```

val init : context * hyp -> proof      (* init *)
val trueR : context -> proof            (* TR *)
val falseL : hyp * sequent -> proof     (* LL *)
val conjR : proof * proof -> proof      (* AR *)
val conjL1 : hyp * proof -> proof       (* AL1 *)
val conjR2 : hyp * proof -> proof       (* AL2 *)
val disjR1 : proof * prop -> proof      (* VR1 *)
val disjR2 : prop * proof -> proof      (* VR2 *)
val disjL : hyp * proof * proof -> proof (* VL *)
val implR : proof -> proof              (* DR *)
val implL : hyp * proof * proof -> proof (* DL *)
end

```

Next, we need to *implement* this signature as structure. To do this, we declare a structure called Kernel which **opaquely** ascribes the signature KERNEL; opaque ascription, written using `:>` below, ensures that the implementation of the type proof remains abstract, i.e. no client of the Kernel structure can see its actual concrete representation. This is what ensures that only the kernel needs to be verified and trusted!

```

structure Kernel :> KERNEL =
struct

```

At this point, we have to decide on an internal representation of proofs. We could choose a *transient* representation, in which the proof trace is forgotten:

```

type proof = sequent
fun infer s = s

```

Then, the implementations of the rules would simply check that their parameters and premises have the right form, and raise an exception if they do not. For instance:

```

(*  $\overline{\Gamma, A, \Delta \implies A}$  init *)
fun init (ctx, i) =
  ctx ===> List.nth (ctx, i)

(*  $\overline{\Gamma \implies \top}$  TR *)
fun trueR ctx =
  ctx ===> TRUE

(*  $\overline{\Gamma, \perp, \Delta \implies a}$   $\perp L$  *)
fun falseL (i, ctx ===> a) =
  if List.nth (ctx, i) = FALSE then
    ctx ===> a
  else
    raise Fail "falseL not applicable"

(*  $\frac{\Gamma \implies A \quad \Gamma \implies B}{\Gamma \implies A \wedge B}$   $\wedge R$  *)
fun conjR (ctx1 ===> a1, ctx2 ===> a2) =
  if ctx1 = ctx2 then
    ctx1 ===> a1 /\ a2
  else
    raise Fail "conjR not applicable"

(*  $\frac{\Gamma, A \wedge B, \Delta, A \implies C}{\Gamma, A \wedge B, \Delta \implies C}$   $\wedge L_1$  *)
fun conjL1 (i, ctx ===> c) =
  case ctx of
    a :: ctx' =>
      (case List.nth (ctx', i) of
        a' /\ b =>
          if a = a' then
            ctx' ===> c
          else
            raise Fail "conjL1 not applicable"
        | _ => raise Fail "conjL1 not applicable")
    | _ => raise Fail "conjL1 not applicable"

(* and so on *)

```

Now, a cleaner and more robust way to write the above rule is the following:

```

(*  $\frac{\Gamma, A \wedge B, \Delta, A \implies C}{\Gamma, A \wedge B, \Delta \implies C}$   $\wedge L_1$  *)
fun conjL1 (i, ctx ===> c) =
  let
    val a :: ctx' = ctx
    val a' /\ b = List.nth (ctx', i)

```

```

    val true = a = a'
in
  ctx' ===> c
end
handle _ => raise Fail "conjL1 not applicable"

```

This pattern is also applicable to the other rules of inference. We leave the implementation of the remaining rules as an exercise.

```
end
```

## 2.1 Evidence-Producing Kernels

The kernel described in the previous section is sufficient for developing and checking sequent calculus proofs. For instance, consider the following sequent calculus derivation:

$$\frac{\overline{A \wedge B, A \implies A}}{A \wedge B \implies A} \text{ init}$$

This is encoded in our kernel as follows:

```

structure K = Kernel
val d : K.proof = K.conjL1 (0, K.init ([`"A", `"A" /\ `"B"], 0))

```

However, the proof object `d` above does not actually contain any information about how the proof was derived; it may be more accurate to call it a “proof certificate” than to call it a “proof”. If we wish to be able to inspect the proof derivation after it has been constructed, we may provide a different implementation of the `KERNEL` signature where the type `proof` is implemented by some kind of proof tree.

However, if we do this, then we lose abstraction: someone else could easily produce such a proof tree outside of our kernel. How can we cleanly achieve both abstraction and inspectability of proofs? One approach is to use a *view* together with an abstract type.

Let us begin by defining a type parametric in some type variable `'a`, which captures the shape of sequent calculus proof trees, but allows the subtrees to be implemented by `'a`.

```

type hyp = int

datatype 'a deriv =
  INIT of hyp
| TRUE_R
| FALSE_L of hyp
| CONJ_R of 'a * 'a
| CONJ_L1 of hyp * 'a
| CONJ_L2 of hyp * 'a
| DISJ_R1 of 'a
| DISJ_R2 of 'a

```

```

| DISJ_L of hyp * 'a * 'a
| IMPL_R of 'a
| IMPL_L of hyp * 'a * 'a

```

The idea is that the position of subtrees in each derivation rule are replaced with '`'a`'. Now, we can interleave the abstract proof type with the type of derivations by supplying it for '`'a`' in the following way:

```

structure EvidenceKernel :>
sig
  include KERNEL
  val unroll : proof -> proof deriv
end =
struct
  datatype proof = BY of sequent * proof deriv
  infix BY

  fun infer (s BY _) = s
  fun unroll (_ BY m) = m

  fun init (ctx, i) =
    ctx ===> List.nth (ctx, i) BY INIT i
    handle _ => raise Fail "init not applicable"

  fun trueR ctx =
    ctx ===> TRUE BY TRUE_R

  fun falseL (i, ctx ===> p) =
    let
      val FALSE = List.nth (ctx, i)
    in
      ctx ===> p BY (FALSE_L i)
    end
    handle _ => raise Fail "falseL not applicable"

  fun conjR (d1 as ctx1 ===> p1 BY _, d2 as ctx2 ===> p2 BY _) =
    let
      val true = ctx1 = ctx2
    in
      ctx1 ===> (p1 /\ p2) BY CONJ_R (d1, d2)
    end
    handle _ => raise Fail "conjR not applicable"

  fun conjL1 (i, d as ctx ===> r BY _) =
    let
      val p :: ctx' = ctx

```

```

    val p' /\ q = List.nth (ctx', i)
    val true = p = p'
in
  ctx' ==> r BY CONJ_L1 (i, d)
end
handle _ => raise Fail "conjL1 not applicable"

(* and so on *)
end

```

**Exercise 2.1.** Now construct a **SML** function to pretty print the derivation that corresponds to a value of type `EvidenceKernel.proof`, using `EvidenceKernel.unroll` and structural recursion.

```

fun pretty (d : proof) : string =
  raise Fail "TODO"

```

### 3 Refinement Proof and Backward Inference

It is often frustrating to construct proofs manually using the primitives that are exposed by a forward inference kernel `K:KERNEL`. Informally, sequent calculus is optimized for upward (backward) inference from conclusion to premises; the kernel seems to force us to perform proofs inside-out. When using the kernel, it is also necessary to pass annoying parameters, such as the context parameter in `K.trueR`.

Separately for any such kernel, we can develop what is called a *refiner*, which is a module that allows us to construct proofs from the bottom up, without needing to pass in any unnecessary parameters. Regarded as a component of a proof system, because the refiner ultimately is a mode of use of the kernel, it does not need to be trusted.

In the context of refinement proof, we will use the word “goal” to mean a sequent. A *refinement rule* is a partial function that assigns to some goal a list of subgoals, together with a *validation*. The input goal correspond to the *conclusion* of a sequent calculus rule, and the subgoals correspond to the premises. A *validation* is a function that takes a list of proof objects (proofs of the premises) and constructs a new proof object (a proof of the conclusion). Validations are always constructed using the forward inference rules exposed by the kernel.

In **SML**, these concepts are rendered as follows:

```

type goal = sequent
type subgoals = goal list
type validation = K.proof list -> K.proof
type rule = goal -> subgoals * validation

```

A completed refinement proof produces the empty list of subgoals; therefore, its validation can be instantiated with the empty list of proofs, yielding a proof of the main conclusion.

We will implement the refiner as a structure *fibered* over a kernel `K`:

```

signature REFINER =
sig
  structure K : KERNEL
  type goal = sequent
  type subgoals = goal list
  type validation = K.proof list -> K.proof
  type rule = goal -> subgoals * validation

  val init : hyp -> rule
  val trueR : rule
  val falseL : hyp -> rule
  val conjR : rule
  val conjL1 : hyp -> rule
  val conjL2 : hyp -> rule
  val disjR1 : rule
  val disjR2 : rule
  val disjL : rule
  val implR : rule
  val implL : rule
end

```

Such a signature is implemented via a *functor* from any kernel K:

```

functor Refiner (K : KERNEL) : REFINER =
struct
  structure K = K
  type goal = sequent
  type subgoals = goal list
  type validation = K.proof list -> K.proof
  type rule = goal -> subgoals * validation

```

Now observe how we implement the backward inference version of init. The input to our function is the *conclusion* of the rule, and we check that the side conditions are satisfied; then we return the empty list of subgoals (there were no premises), and for the validation, we call K.init from the kernel.

```

fun init i (ctx ===> a) =
  let
    val true = List.nth (ctx, i) = a
  in
    ([] , fn [] => K.init (ctx, i))
  end
  handle _ => raise Fail "init not applicable"

```

The next two rules follow a similar pattern and are not very interesting.

```

fun trueR (ctx ===> a) =
  let

```

```

    val TRUE = a
in
  ([] , fn [] => K.trueR ctx)
end
handle _ => raise Fail "trueR not applicable"

fun falseL i (ctx ===> a) =
  let
    val FALSE = List.nth (ctx, i)
  in
    ([] , fn [] => K.falseL (i, ctx ===> a))
  end
handle _ => raise Fail "falseL not applicable"

```

The rules for conjunction are a bit more illustrative:

```

fun conjR (ctx ===> r) =
  let
    val p /\ q = r
  in
    ([ctx ===> p, ctx ===> q],
     fn [d1, d2] => K.conjR (d1, d2))
  end
handle _ => raise Fail "conjR not applicable"

fun conjL1 i (ctx ===> r) =
  let
    val p /\ _ = List.nth (ctx, i)
  in
    ([p :: ctx ===> r],
     fn [d] => K.conjL1 (i, d))
  end
handle _ => raise Fail "conjL1 not applicable"

```

We leave the remainder of the refinement rules as an exercise.

```

end

structure R = Refiner (EvidenceKernel)

```

## 4 Tactics: combinators for refinement rules

It is hard to see how to use refinement rules to construct proofs on their own. However, there are a number of well-known combinators for refinement rules which correspond to *derived rules* in sequent calculus; in the **LCF** tradition, derived rules are called *tactics*, and the combinators from which they are built are called *tacticals* (by analogy with *functionals*).

```

signature TACTIC =
sig
  structure R : REFINER
  type tactic = R.rule

  val thenl : tactic * tactic list -> tactic
  val then_ : tactic * tactic -> tactic
  val orelse_ : tactic * tactic -> tactic
end

```

The most fundamental tactical is `thenl`; the tactic `thenl` (`t`, `ts`) uses the tactic `t` to decompose the current goal into  $n$  subgoals; then, the list of tactics `ts` (also of length  $n$ ) is applied *pointwise* to further decompose these subgoals. Then, the resulting lists of subgoals are all combined into a single list, which is returned together with a validation that performs essentially the inverse process for forward inference. The tactical `then_` is similar, except it uses its second argument to decompose *all* of the remaining subgoals.

Finally, the tactic `orelse_` (`t1`, `t2`) tries to decompose the current goal with `t1`; if this fails, it continues with `t2`.

## 4.1 Implementing tactics

The implementation of the standard tactics above is provided below as a reference; it relies on some tricky list manipulation, but the good news is you only need to implement it once.

```

functor Tactic (R : REFINER) : TACTIC =
struct
  structure R = R
  open R

  type tactic = goal -> subgoals * validation

  fun splitAt (xs, i) =
    let
      val front = List.take (xs, i)
      val back = List.drop (xs, i)
    in
      (front, back)
    end

  fun gobbleWith ([] , []) args = []
    | gobbleWith (n :: ns, f :: fs) args =
      let
        val (xs, args') = splitAt (args, n)
      in

```

```

        f xs :: gobbleWith (ns, fs) args'
    end

fun stitchProof (validation, subgoalss, validations) =
  (List.concat subgoalss,
   validation o
   gobbleWith (map length subgoalss, validations))

fun then_ (t1, t2) goal =
  let
    val (subgoals, validation) = t1 goal
    val (subgoalss, validations) =
      ListPair.unzip (List.map t2 subgoals)
  in
    stitchProof (validation, subgoalss, validations)
  end

fun thenl (t, ts) goal =
  let
    val (subgoals, validation) = t goal
    val (subgoalss, validations) =
      ListPair.unzip
      (ListPair.mapEq
       (fn (t, g) => t g)
       (ts, subgoals))
  in
    stitchProof (validation, subgoalss, validations)
  end

fun orelse_ (t1, t2) (goal : goal) =
  t1 goal handle _ => t2 goal
end

structure T = Tactic (R)
open T infix then_ thenl orelse_

```

Now, we can use tactics to capture the backward-inference version of the following proof:

$$\frac{\frac{\frac{A \wedge B, A, B \implies B \text{ init} \quad A \wedge B, A, B \implies A \text{ init}}{A \wedge B, A, B \implies B \wedge A} \wedge R}{A \wedge B, A \implies B \wedge A} \wedge L_2}{A \wedge B \implies B \wedge A} \wedge L_1$$

```

val t : tactic =
  R.conjL1 0 then_

```

```

R.conjL2 1 then_
R.conjR thenl
[R.init 0,
 R.init 1]

val result = t ([`"A" /\ ``B"] ==> ``B" /\ ``A")
val d : proof = #2 result []

```

## 4.2 Possible extensions

**LCF-style tactics** do not satisfy every need; many different extensions are possible.

**Existential variables and unification** It is difficult to capture a usable refinement proof theory for logics with existential quantifiers using pure **LCF**; rather than having a single introduction rule for the existential quantifier, it is necessary to have a countable family of such introduction rules, parameterized in the actual witness of the existential. This is highly disruptive to the refinement proof process, since it may be that one only determines how to instantiate the existential  $\exists x.A(x)$  by attempting to prove the predicate  $A$ .

To resolve this contradiction, most modern proof assistants add a notion of existential variable, which allows one to decompose the goal  $\exists x.A(x)$  into  $A(?x)$ ; then, later on in this subproof, the variable  $?x$  can be instantiated by *unification* with something concrete (like  $?x := 42$ ).

Existential variables introduce many complexities into the design of **LCF**-style proof assistants, partly because of the difficulty (and in some cases, impossibility) of finding most-general unifiers. In **Coq** [10], a higher-order unification algorithm is used which produces unifiers which are not most general [11], but because **Coq** adheres to the **LCF** architecture, this only affects the ergonomics of the tactic system as opposed to the core logic. Additionally, existential variables disrupt the *local* character of **LCF**-style proof refinement: every refinement step can affect the whole proof state.

On the other hand, higher-order unification is built into the trusted kernel of **Isabelle**, which uses both *dynamic pattern unification* (which produces most general unifiers at the cost of being somewhat restrictive) and general higher-order unification, which may produce infinitely many unifiers (or none).

One benefit of building unification into the core is that it is possible to simplify the notion of proof refinement significantly, capturing both forward and backward inference in a single kernel [5]; in turn, this obviates the notion of *validation*, which is the hardest part of **LCF**-style tactic systems to implement correctly.

**Backtracking** The `orelse_` tactical enables a form of proof search procedure, but it cannot be used to implement backtracking. There are at least two ways to extend **LCF** with support for backtracking; one way would be using continuations, but the most common way is to replace the type of tactics with something that returns a (lazy) sequence of proof states as follows:

```
type tactic = R.goal -> (R.subgoals * R.validation) Seq.seq
```

Then, a backtracking tactical `par : tactic * tactic -> tactic` will simply merge the results of applying *both* tactics into a single sequence:

```
fun par (t1, t2) goal =
  Sequence.merge (t1 goal, t2 goal)
```

Support for backtracking is useful in any proof assistant, but becomes absolutely essential in the presence of existential variables. Backtracking using sequences of results was first introduced in **Isabelle**.

**Dependent refinement** In the context of implementing dependent type theory, it is useful to consider a notion of refinement rule in which the *statement* of one premise may refer to the *proof* of a previous premise. This is best considered a separate issue from the matter of existential variables, but concrete implementations of this behavior may either use existential variables (as in [7, 1]) or not (as in [8, 9]).

## References

- [1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 64–69, 2011.
- [2] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1979.
- [3] Mike Gordon. From LCF to HOL: A short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.
- [4] Lawrence C. Paulson. *Logic and computation : interactive proof with Cambridge LCF*. Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge, New York, Port Chester, 1987. Autre tirage : 1990 (br.).
- [5] Lawrence C. Paulson. Strategic principles in the design of isabelle. In *In CADE-15 Workshop on Strategies in Automated Deduction*, pages 11–17, 1998.
- [6] Robert Pollack. On extensibility of proof checkers. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs: International Workshop TYPES '94 Båstad, Sweden, June 6–10, 1994 Selected Papers*, pages 140–161, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [7] Arnaud Spiwack. *Verified Computing in Homological Algebra, A Journey Exploring the Power and Limits of Dependent Type Theory*. PhD thesis, École Polytechnique, 2011.
- [8] Jonathan Sterling and Robert Harper. Algebraic foundations of proof refinement. <https://arxiv.org/abs/1703.05215>, 2017.

- [9] Jonathan Sterling, Kuen-Bang Hou (Favonia), Daniel Gratzer, David Christiansen, Darin Morrison, Eugene Akentyev, and James Wilcox. RedPRL – the People’s Refinement Logic. <http://www.redprl.org/>, 2016.
- [10] The Coq Development Team. The Coq Proof Assistant Reference Manual, 2016.
- [11] Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *20th ACM SIGPLAN International Conference on Functional Programming*, Vancouver, Canada, September 2015.

# Recitation 8: Bidirectional Typechecking and Sequent Calculus

Jon Sterling

## 1 Bidirectional Typechecking

The proof term assignment for the logic of verifications and uses can be construed as an algorithmic specification for a typechecker; this is because the rules of this logic are all *syntax-directed*.

To expose the algorithmic character of the judgments, we write in blue the *inputs* of a judgment and in red the *outputs* of a judgment.

$$\begin{array}{c}
 \frac{}{() : \top \uparrow} \text{TI} \quad \frac{R : \perp \downarrow}{\text{abort}(R) : C \uparrow} \perp E \quad \frac{M : A \uparrow \quad N : B \uparrow}{(M, N) : A \wedge B \uparrow} \wedge I \\
 \\ 
 \frac{R : A \wedge B \downarrow}{\text{fst}(R) : A \downarrow} \wedge E_1 \quad \frac{R : A \wedge B \downarrow}{\text{snd}(R) : B \downarrow} \wedge E_2 \quad \frac{M : A \uparrow}{\text{inl}(M) : A \vee B \uparrow} \vee I_1 \\
 \\ 
 \frac{M : B \uparrow}{\text{inr}(M) : A \vee B \uparrow} \vee I_2 \quad \frac{\begin{array}{c} R : A \vee B \downarrow \quad N_1 : C \uparrow \quad N_2 : C \uparrow \\ \text{case } R \text{ of inl}(u) \Rightarrow N_1 \mid \text{inr}(v) \Rightarrow N_2 : C \uparrow \end{array}}{u : A \downarrow \quad v : B \downarrow} \vee E^{u,v} \\
 \\ 
 \frac{\begin{array}{c} u : A \downarrow \\ \vdots \\ N : B \uparrow \end{array}}{\text{fn } u \Rightarrow N : A \supset B \uparrow} \supset I^u \quad \frac{R : A \supset B \downarrow \quad N : A \uparrow}{R(N) : B \downarrow} \supset E \\
 \\ 
 \frac{R : A \downarrow}{R : A \uparrow} \downarrow \uparrow \quad \frac{M : A \uparrow}{(M : A) : A \downarrow} \text{ann}
 \end{array}$$

If the last rule *ann* is omitted, then the terms of the verifications and uses calculus will have no redexes; if it is added, then redexes can be formed, and the calculus is merely a more verbose version of the ordinary  $\lambda$ -calculus. This week, we are studying a version of verifications and uses with the *ann* rule, because it is necessary in order to give a natural proof term assignment to Dyckhoff's contraction-free sequent calculus.

$$\Delta \vdash \dots$$

**the algorithm** We will use  $\Delta \vdash \mathcal{J}$  to abbreviate  $\vdash \mathcal{J}$  in what follows, where  $\Delta$  is a sequence of assumptions  $x : A \downarrow$ . The typechecking algorithm for bidirectional type-checking is the computational content of the following (constructive) metatheorem:

1. For all  $\Delta, N, A$ , either  $\Delta \vdash N : A \uparrow$  or not.
2. For all  $\Delta, R$ , either there exists some  $A$  such that  $\Delta \vdash R : A \downarrow$  or there does not.

**Exercise.** Try and remember how this proof works. You will have to extract its algorithmic content in order to complete the next homework assignment.

**Remark.** This metatheorem is only interesting if the ambient mathematics is constructive; otherwise, it is trivial and its proof provides no useful algorithmic content.

## 2 Proof terms for sequent calculus

We can give a proof term assignment to the sequent calculus based on a single form of judgment  $\Gamma \Rightarrow N : A$ ; in this form of judgment,  $\Gamma$  is now a list of formal type assignments  $R : A$ , where  $R$  is an arbitrary neutral term rather than only a variable. The dynamics are as follows:

1. The list of type assignments  $\Gamma \equiv \overline{R : A}$  are inputs.
2. The goal type  $A$  is an input.
3. The proof term / witness  $N$  is an output.

These dynamics correspond to *proof refinement with extraction* in sequent-calculus-based proof systems like **Nuprl** and **RedPRL**.

$$\begin{array}{c}
\frac{}{\Gamma, R : A \Rightarrow R : A} \text{ init} \quad \frac{}{\Gamma \Rightarrow () : \top} \text{ TR} \quad \frac{}{\Gamma, R : \perp \Rightarrow \text{abort}(R) : C} \perp L \\
\frac{\Gamma \Rightarrow N_1 : A}{\Gamma \Rightarrow (N_1, N_2) : A \wedge B} \wedge R \quad \frac{\Gamma, R : A \wedge B, \text{fst}(R) : A \Rightarrow N : C}{\Gamma, R : A \wedge B \Rightarrow N : C} \wedge L_1 \\
\frac{\Gamma, R : A \wedge B, \text{snd}(R) : B \Rightarrow N : C}{\Gamma, R : A \wedge B \Rightarrow N : C} \wedge L_2 \quad \frac{\Gamma \Rightarrow N : A}{\Gamma \Rightarrow \text{inl}(N) : A \vee B} \vee R_1 \\
\frac{\Gamma \Rightarrow N : B}{\Gamma \Rightarrow \text{inr}(N) : A \vee B} \vee R_2 \\
\frac{\Gamma, R : A \vee B, u : A \Rightarrow N_1 : C \quad \Gamma, R : A \vee B, v : A \Rightarrow N_2 : C}{\Gamma, R : A \vee B \Rightarrow \text{case } R \text{ of inl}(u) \Rightarrow N_1 \mid \text{inr}(v) \Rightarrow N_2 : C} \vee L^{u,v}
\end{array}$$

$$\frac{\Gamma, u : A \Rightarrow N : B}{\Gamma \Rightarrow \text{fn } u \Rightarrow N : A \supset B} \supset R^u$$

$$\frac{\Gamma, R : A \supset B \Rightarrow M : A \quad \Gamma, R : A \supset B, R(M) : B \Rightarrow N : C}{\Gamma, R : A \supset B \Rightarrow N : C} \supset L$$

**Exercise.** In sequent calculus, we try to prove the admissibility of the following cut rule:

$$\frac{\Gamma \Rightarrow A \quad \Gamma, A \Rightarrow B}{\Gamma \Rightarrow B} \text{ cut}$$

Try to invent a proof term assignment for this rule.

**Solution.**

$$\frac{\Gamma \Rightarrow M : A \quad \Gamma, (M : A) : A \Rightarrow N : B}{\Gamma \Rightarrow N : B} \text{ cut}_A$$

# Constructive Logic (15-317), Fall 2017

## Recitation 9: Logic programming

October 26, 2017

### 1 Logic programming

You might be familiar with functional and imperative programming. Today we will see yet another programming paradigm: logic programming. Logic programming can be seen as a fragment of intuitionistic logic<sup>1</sup> called *Horn clauses* (remember last homework?). A Horn clause is either an atom or a formula of the shape  $A_1 \wedge \dots \wedge A_n \supset H$ , where  $H$  is called the *head* and  $A_1 \wedge \dots \wedge A_n$  is the *body*. In prolog syntax, this is written as:

```
h :- a1, a2, ..., an.
```

Let's step through a simple prolog program to understand how computation (or proof search) works. Consider the following simple code:

```
ocean_level(rising).  
temperature(extreme).  
global_warming(conspiracy) :- ocean_level(stable), temperature(normal).  
global_warming(real) :- ocean_level(rising), temperature(extreme).
```

If we query prolog for `global_warming(X)`, it will look at the head of all (four) clauses trying to find one that "matches" (*unifies*) with the goal. In this case, it finds the clauses in lines 3 and 4. Prolog will process the options in order, so it will first go to clause in line 3 and unify `X` with `conspiracy`, generating the new goals `ocean_level(stable)` and `temperature(normal)`. A proof-theoretic interpretation of this step is the following (predicate names are abbreviated for the sake of space):

$$\frac{\text{ol(ris), temp(xtr), ...} \longrightarrow \text{ol(sta)} \quad \text{ol(ris), temp(xtr), ...} \longrightarrow \text{temp(nml)}}{\text{ol(ris), temp(xtr), ...} \longrightarrow \text{ol(sta)} \wedge \text{temp(nml)}} \wedge R \quad \frac{\text{x is csp}}{\text{ol(ris), temp(xtr), gw(csp), ...} \longrightarrow \text{gw(X)}} \text{ init} \\ \text{ol(ris), temp(xtr), ol(sta) \wedge temp(nml)} \supset \text{gw(csp)}, \text{ol(ris) \wedge temp(xtr)} \supset \text{gw(real)} \longrightarrow \text{gw(X)} \supset L$$

In this derivation, `X` is a special variable that is unified on initial rules, and this unification propagates to the next branch if there were occurrences of `X` there as well. When trying to

---

<sup>1</sup>It is also a fragment of classical logic. Since it is such a simple fragment, intuitionistic and classical logic coincide.

prove the two open sequents, or the new goals, prolog will realize that `ocean_level(stable)` or `temperature(normal)` are not true... oops, are not in the context nor they are unifiable with any clause head. Time to backtrack. We know that  $\wedge R$  is an invertible rule, so no use in backtracking there. We go back to the choice of clauses (i.e.,  $\supset L$ ) and try to use the one on line 4. This time the unification will be `X is real` and the new goals `ocean_level(rising)` and `temperature(extreme)`, which can be proved.

As a final note, logic programs hold some resemblance to functional programs in the way programs are written. You will find that sometimes the clauses used look a lot like the cases you would need in, say, SML. This kind of programming style is referred to as *declarative* programming (you write *what* your program does as opposed to *how* it does it).

**Task 1.** Implement a prolog program that computes the truncated subtraction between natural number along the same lines as the plus and times implementations given in the lecture notes.

```
pred(z, z).

pred(s(M), M).

minus(N, z, N).

minus(N, s(M), Q) :- minus(N, M, P), pred(P, Q).
```

In Prolog, lists are built in similarly to SML. The syntax for pattern matching on a list is `[Head | Tail]`. Using this we can implement a variety of programs for manipulating lists.

**Task 2.** Implement a prolog program which merges two sorted lists.

```
mymerge([], [], L).
mymerge([], L, L).
mymerge([H1 | T1], [H2 | T2], [H1 | Out]) :-
    H1 =< H2,
    mymerge(T1, [H2 | T2], Out).
mymerge([H1 | T1], [H2 | T2], [H2 | Out]) :-
    H1 > H2,
    mymerge([H1 | T1], T2, Out).
```

**Task 3.** Implement a merge sorting procedure for lists.

```
split([], [], []).
split([X], [X], []).
split([H1 | [H2 | T]], [H1 | L1], [H2 | L2]) :-
    split(T, L1, L2).

mysort([]).
mysort([X]).
mysort([X1 | [X2 | L]], O) :-
    split([X1 | [X2 | L]], Left, Right),
    mysort(Left, SLeft),
    mysort(Right, SRight),
    mymerge(SLeft, SRight, O).
```

## 2 Modes

It's common in Prolog code to denote certain arguments to a relation as "inputs" and some as "outputs". These is the *mode* of an argument. An important property to ensure that your prolog programs terminate is to ensure that they are well-moded. That is, the inputs to a subgoal as well as the outputs are either determined by inputs or outputs of a previous goal. For instance, attempt to verify whether or not the following prolog programs are well-moded.

```
notprime(P) :-  
    divisible(P, Q) %% divisible takes two inputs and holds when P \% Q = 0  
  
times(z, N, z).  
times(s(M), N, O) :-  
    times(M, N, U),  
    plus(U, N, O).  
  
% recall synth has the mode input, output  
synth(inl(M), or(A, B)) :-  
    synth(M, A).
```