Lecture Notes on Constructive Logic: Overview

15-317: Constructive Logic
Frank Pfenning

Lecture 1
August 25, 2009

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. Proofs as Computations

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[1], 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 finite-state 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 introduction to this software is included in these notes, a separate manual is available with the on-line course material.

[1] for example: classical, intuitionistic, modal, second-order, temporal, belief, linear, relevance, affirmation, ...

Lecture Notes on Natural Deduction

15-317: Constructive Logic
Frank Pfenning

Lecture 2
August 27, 2009

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.

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. The general form of an inference rule is

$$\frac{J_1 \ \ldots \ J_n}{J} \ \textit{name}$$

where the judgments $J_1, \ldots, J_n$ 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 deduce 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:

$$\frac{A \wedge B \ \mathit{true}}{A \ \mathit{true}} \wedge E_L \qquad\qquad \frac{A \wedge B \ \mathit{true}}{B \ \mathit{true}} \wedge E_R$$

The name $\wedge E_L$ stands for "left conjunction elimination", since the conjunction in the premise has been eliminated in the conclusion. Similarly $\wedge E_R$ stands for "right conjunction elimination".

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

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

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 some arbitrary propositions A, B, and C:

$$\dfrac{\dfrac{A \wedge (B \wedge C) \ \mathit{true}}{B \wedge C \ \mathit{true}} \wedge E_R}{B \ \mathit{true}} \wedge E_L$$

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

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

where the judgments $J_1, \ldots, J_n$ are unproven assumptions, and the judgment J is the conclusion. Note that we can always substitute a proof for any hypothesis J_i to eliminate the assumption. We call this the *substitution principle* for hypotheses.

Many mistakes in reasoning arise because dependencies on some hidden assumptions are ignored. When we need to be explicit, we write $J_1, \ldots, J_n \vdash J$ for the hypothetical judgment which is established by the hypothetical derivation above. We may refer to $J_1, \ldots, J_n$ as the antecedents and J as the succedent of the hypothetical judgment.

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,

$$\dfrac{\dfrac{A \wedge B \ \mathit{true}}{B \ \mathit{true}}\,\wedge E_R \qquad \dfrac{A \wedge B \ \mathit{true}}{A \ \mathit{true}}\,\wedge E_L}{B \wedge A \ \mathit{true}}\,\wedge I$$

can be seen a hypothetical derivation of $A \wedge B \ \mathit{true} \vdash B \wedge A \ \mathit{true}$.

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.

$$\dfrac{\begin{array}{c}\overline{A \ \mathit{true}}\ u \\ \vdots \\ B \ \mathit{true}\end{array}}{A \supset B \ \mathit{true}}\,\supset I^u$$

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

$$\dfrac{\begin{array}{c} A\ \mathit{true} \\ \vdots \\ B\ \mathit{true} \end{array}}{A \supset B\ \mathit{true}}\ \supset I$$

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! 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.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A\ \mathit{true}}\,u \quad \dfrac{}{B\ \mathit{true}}\,w}{A \wedge B\ \mathit{true}}\ \wedge I}{B \supset (A \wedge B)\ \mathit{true}}\ \supset I^w}{A \supset (B \supset (A \wedge B))\ \mathit{true}}\ \supset I^u$$

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*.

$$\dfrac{A \supset B\ \mathit{true} \quad A\ \mathit{true}}{B\ \mathit{true}}\ \supset E$$

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{A \wedge B\ \mathit{true}}\ u}{B\ \mathit{true}}\wedge E_R \quad \dfrac{\overline{A \wedge B\ \mathit{true}}\ u}{A\ \mathit{true}}\wedge E_L}{B \wedge A\ \mathit{true}}\wedge I}{(A \wedge B) \supset (B \wedge A)\ \mathit{true}}\supset I^u$$

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.

$$\begin{array}{c}\vdots\\ (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}\end{array}$$

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

$$\dfrac{\begin{array}{c}\overline{A \supset (B \wedge C)\ \mathit{true}}\ u\\ \vdots\\ (A \supset B) \wedge (A \supset C)\ \mathit{true}\end{array}}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\supset I^u$$

Next, we use the conjunction introduction rule bottom-up.

$$\dfrac{\dfrac{\begin{array}{c}\overline{A \supset (B \wedge C)\ \mathit{true}}\ u\\ \vdots\\ A \supset B\ \mathit{true}\end{array} \quad \begin{array}{c}\overline{A \supset (B \wedge C)\ \mathit{true}}\ u\\ \vdots\\ A \supset C\ \mathit{true}\end{array}}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\wedge I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\supset I^u$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{\vdots}\supset E}{\dfrac{B\ \mathit{true}}{A \supset B\ \mathit{true}}\supset I^w} \qquad \dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u}{\vdots}}{\dfrac{(A \supset B) \wedge (A \supset C)\ \mathit{true}}{A \supset C\ \mathit{true}}\wedge I}}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\supset I^u$$

Note that the hypothesis $A\ \mathit{true}$ is available only in the left branch, but 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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{B \wedge C\ \mathit{true}}\supset E}{\vdots}}{\dfrac{B\ \mathit{true}}{A \supset B\ \mathit{true}}\supset I^w} \qquad \dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u}{\vdots}}{\dfrac{(A \supset B) \wedge (A \supset C)\ \mathit{true}}{A \supset C\ \mathit{true}}\wedge I}}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\supset I^u$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{B \wedge C\ \mathit{true}}\supset E}{B\ \mathit{true}}\wedge E_L}{A \supset B\ \mathit{true}}\supset I^w \qquad \dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u}{\vdots}}{\dfrac{(A \supset B) \wedge (A \supset C)\ \mathit{true}}{A \supset C\ \mathit{true}}\wedge I}}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\supset I^u$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{B \wedge C\ \mathit{true}}\supset E}{B\ \mathit{true}}\wedge E_L}{A \supset B\ \mathit{true}}\supset I^w \quad \dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,v}{B \wedge C\ \mathit{true}}\supset E}{C\ \mathit{true}}\wedge E_R}{A \supset C\ \mathit{true}}\supset I^v}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\wedge I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\supset I^u$$

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.

$$\dfrac{A\ \mathit{true}}{A \vee B\ \mathit{true}}\vee I_L \qquad \dfrac{B\ \mathit{true}}{A \vee B\ \mathit{true}}\vee I_R$$

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

$$\dfrac{A \vee B\ \mathit{true}}{A\ \mathit{true}}\vee E_L?$$

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:

$$\dfrac{\dfrac{\dfrac{}{\top\ \mathit{true}}\top I}{A \vee \top\ \mathit{true}}\vee I_R}{A\ \mathit{true}}\vee E_L?$$

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:

$$\dfrac{A \vee B\ \mathit{true} \qquad \begin{array}{c}\dfrac{}{A\ \mathit{true}}\,u\\ \vdots \\ C\ \mathit{true}\end{array} \qquad \begin{array}{c}\dfrac{}{B\ \mathit{true}}\,w\\ \vdots \\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\vee E^{u,w}$$

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.

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

We begin with an implication introduction.

$$\dfrac{\dfrac{}{A \vee B \text{ true}}\,u \quad \vdots \quad B \vee A \text{ true}}{(A \vee B) \supset (B \vee A) \text{ true}}\,\supset I^u$$

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.

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

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.

$$\dfrac{\dfrac{}{A \vee B\ \mathit{true}}\,u \quad \dfrac{\dfrac{}{A\ \mathit{true}}\,v}{B \vee A\ \mathit{true}}\vee I_R \quad \dfrac{\dfrac{}{B\ \mathit{true}}\,w}{B \vee A\ \mathit{true}}\vee I_L}{\dfrac{B \vee A\ \mathit{true}}{(A \vee B) \supset (B \vee A)\ \mathit{true}}\supset I^u}\vee E^{v,w}$$

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

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

$$\frac{\bot\ \mathit{true}}{C\ \mathit{true}}\,\bot E$$

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 $\bot E$ rule above.

From this is might seem that falsehood it 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 $A \supset \bot$. In other words, $\neg A$ is true precisely if the assumption A *true* is contradictory because we could derive $\bot$ *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	**Elimination Rules**
$\dfrac{A\ true \quad B\ true}{A \wedge B\ true}\ \wedge I$	$\dfrac{A \wedge B\ true}{A\ true}\ \wedge E_L \quad \dfrac{A \wedge B\ true}{B\ true}\ \wedge E_R$
$\dfrac{}{\top\ true}\ \top I$	*no* $\top E$ *rule*
$\dfrac{\begin{array}{c}\overline{A\ true}\ u \\ \vdots \\ B\ true\end{array}}{A \supset B\ true}\ \supset I^u$	$\dfrac{A \supset B\ true \quad A\ true}{B\ true}\ \supset E$
$\dfrac{A\ true}{A \vee B\ true}\ \vee I_L \quad \dfrac{B\ true}{A \vee B\ true}\ \vee I_R$	$\dfrac{A \vee B\ true \quad \begin{array}{c}\overline{A\ true}\ u \\ \vdots \\ C\ true\end{array} \quad \begin{array}{c}\overline{B\ true}\ w \\ \vdots \\ C\ true\end{array}}{C\ true}\ \vee E^{u,w}$
no $\bot I$ *rule*	$\dfrac{\bot\ true}{C\ true}\ \bot E$

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 $\neg A = (A \supset \bot)$. 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 that 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.

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.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

Lecture Notes on Harmony

15-317: Constructive Logic
Frank Pfenning

Lecture 3
September 1, 2009

1 Introduction

In the verificationist definition of the logical connectives via their introduction rules we have briefly justified the elimination rules. In this section we study the balance between introduction and elimination rules more closely. In order to show that the two 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 the elimination which does not first introduce and then eliminate 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 some eliminations followed by some introductions.

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 are *pure*: the may refer and employ different judgments and judgment forms, but they may

not refer to other propositions 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

$$\begin{array}{c}\mathcal{D} \\ A\ true\end{array} \Longrightarrow_R \begin{array}{c}\mathcal{D}' \\ A\ true\end{array}$$

for the local reduction of a deduction $\mathcal{D}$ to another deduction $\mathcal{D}'$ of the same judgment $A\ true$. In fact, $\Longrightarrow_R$ 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,

$$\begin{array}{c}\mathcal{D} \\ A\ true\end{array} \Longrightarrow_E \begin{array}{c}\mathcal{D}' \\ A\ true\end{array}$$

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

Conjunction. We start with local soundness. Since there are two elimination rules and one introduction, it turns out we have two cases to consider. In either case, we can easily reduce.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D} \\ A\ true\end{array} \quad \begin{array}{c}\mathcal{E} \\ B\ true\end{array}}{A \wedge B\ true}\wedge I}{A\ true}\wedge E_L \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{D} \\ A\ true\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D} \\ A\ true\end{array} \quad \begin{array}{c}\mathcal{E} \\ B\ true\end{array}}{A \wedge B\ true}\wedge I}{B\ true}\wedge E_R \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{E} \\ B\ true\end{array}$$

Local completeness requires us to apply eliminations to an arbitrary proof of $A \wedge B\ true$ in such a way that we can reconstitute a proof of $A \wedge B$ from the results.

$$\begin{array}{c}\mathcal{D} \\ A \wedge B\ true\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D} \\ A \wedge B\ true\end{array}}{A\ true}\wedge E_L \quad \dfrac{\begin{array}{c}\mathcal{D} \\ A \wedge B\ true\end{array}}{B\ true}\wedge E_R}{A \wedge B\ true}\wedge I$$

As an example where local completeness might fail, consider the case where we "forget" the right elimination rule for conjunction. The remaining rule is still locally sound, 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 *true* for any use of a hypothesis A *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

$$\begin{array}{c} \dfrac{}{A\ \mathit{true}}\ u \\ \mathcal{E} \\ B\ \mathit{true} \end{array}$$

is a hypothetical proof of B *true* under the undischarged hypothesis A *true* labelled u, and

$$\begin{array}{c} \mathcal{D} \\ A\ \mathit{true} \end{array}$$

is a proof of A *true* then

$$\begin{array}{c} \dfrac{\mathcal{D}}{A\ \mathit{true}}\ u \\ \mathcal{E} \\ B\ \mathit{true} \end{array}$$

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.

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{A\ \mathit{true}}\ u \\ \mathcal{E} \\ B\ \mathit{true}\end{array}}{A \supset B\ \mathit{true}}\ \supset I^u \qquad \begin{array}{c}\mathcal{D} \\ A\ \mathit{true}\end{array}}{B\ \mathit{true}}\ \supset E \quad \Longrightarrow_R \quad \begin{array}{c}\dfrac{\mathcal{D}}{A\ \mathit{true}}\ u \\ \mathcal{E} \\ B\ \mathit{true}\end{array}$$

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.

$$\begin{array}{c}\mathcal{D}\\ A \supset B\ \mathit{true}\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A \supset B\ \mathit{true}\end{array} \quad \dfrac{}{A\ \mathit{true}}\,u}{B\ \mathit{true}}\,\supset E}{A \supset B\ \mathit{true}}\,\supset I^u$$

Here u must be chosen fresh: it only labels the new hypothesis $A\ \mathit{true}$ 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.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array}}{A \vee B\ \mathit{true}}\,\vee I_L \quad \begin{array}{c}\dfrac{}{A\ \mathit{true}}\,u\\ \mathcal{E}\\ C\ \mathit{true}\end{array} \quad \begin{array}{c}\dfrac{}{B\ \mathit{true}}\,w\\ \mathcal{F}\\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\,\vee E^{u,w} \quad \Longrightarrow_R \quad \begin{array}{c}\dfrac{\begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array}}{}\,u\\ \mathcal{E}\\ C\ \mathit{true}\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ B\ \mathit{true}\end{array}}{A \vee B\ \mathit{true}}\,\vee I_R \quad \begin{array}{c}\dfrac{}{A\ \mathit{true}}\,u\\ \mathcal{E}\\ C\ \mathit{true}\end{array} \quad \begin{array}{c}\dfrac{}{B\ \mathit{true}}\,w\\ \mathcal{F}\\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\,\vee E^{u,w} \quad \Longrightarrow_R \quad \begin{array}{c}\dfrac{\begin{array}{c}\mathcal{D}\\ B\ \mathit{true}\end{array}}{}\,w\\ \mathcal{F}\\ C\ \mathit{true}\end{array}$$

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

$$\dfrac{A \vee B\ \mathit{true}}{A\ \mathit{true}}\,\vee E_L?$$

and, indeed, we would not be able to reduce

$$\dfrac{\dfrac{A \vee B\ \mathit{true}}{B\ \mathit{true}}\,\vee I_R}{A\ \mathit{true}}\,\vee E_L?$$

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

$$\dfrac{\dfrac{\dfrac{}{\top\ \mathit{true}}\,\top I}{\bot \vee \top\ \mathit{true}}\,\vee I_R}{\bot\ \mathit{true}}\,\vee E_L?$$

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

$$\overset{\mathcal{D}}{A \vee B\ \mathit{true}} \Longrightarrow_L \dfrac{\overset{\mathcal{D}}{A \vee B\ \mathit{true}} \quad \dfrac{\dfrac{}{A\ \mathit{true}}\,u}{A \vee B\ \mathit{true}}\,\vee I_L \quad \dfrac{\dfrac{}{B\ \mathit{true}}\,w}{A \vee B\ \mathit{true}}\,\vee I_R}{A \vee B\ \mathit{true}}\,\vee E^{u,w}$$

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 notation: 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.

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

$$\overset{\mathcal{D}}{\top\ \mathit{true}} \Longrightarrow_E \dfrac{}{\top\ \mathit{true}}\,\top I$$

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 $\bot$ *true* so that we can reintroduce a proof of $\bot$ *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.

$$\overset{\mathcal{D}}{\bot\ \mathit{true}} \Longrightarrow_L \dfrac{\overset{\mathcal{D}}{\bot\ \mathit{true}}}{\bot\ \mathit{true}}\,\bot E$$

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 $A \supset \bot$ which is often abbreviated as $\neg A$. In classical reasoning it is even more frequent, due to the rule of proof by contradiction.

2 Verifications

The verificationist point of view on the meaning of a proposition is that it is determined by its *verifications*. Intuitively, a verification should be a proof that only analyzes the constituents of a propositions. This restriction of the space of all possible proofs is necessary so that the definition is well-founded. For example, if in order to understand the meaning of A, we would have to understand ther meaning of $B \supset A$ and B, the whole program of understanding the meaning of the connectives by their proofs 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. Eventually we will also establish the converse: if A is true than A has a verification.

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

$$\dfrac{\dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\ u}{A\ \mathit{true}}\ \wedge E_L}{(A \wedge B) \supset A\ \mathit{true}}\ \supset I^u$$

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:

$$\frac{A\uparrow \quad B\uparrow}{A \wedge B\uparrow}\ \wedge I \qquad \frac{A \wedge B\downarrow}{A\downarrow}\ \wedge E_L \qquad \frac{A \wedge B\downarrow}{B\downarrow}\ \wedge E_R$$

The 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.

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

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

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

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

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

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

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

$$\dfrac{\begin{array}{ccc} & \overline{A\downarrow}\,^{u} & \overline{B\downarrow}\,^{w} \\ & \vdots & \vdots \\ A \vee B\downarrow & C\uparrow & C\uparrow \end{array}}{C\uparrow}\ \vee E^{u,w}$$

Truth. The only verification of truth is the trival one.

$$\dfrac{}{\top\uparrow}\ \top I$$

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.

$$\dfrac{\bot\downarrow}{C\uparrow}\ \bot E$$

Atomic propositions. How to 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.

$$\dfrac{P\downarrow}{P\uparrow}\ \downarrow\uparrow$$

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 than we may use A without gaining any information. That is, for arbitrary propositions A and C:

$$\begin{array}{l} \qquad A{\downarrow} \\ \qquad \vdots \\ \textit{If } A{\uparrow} \textit{ and } C{\uparrow} \textit{ then } C{\uparrow}. \end{array}$$

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 use carry out this proof later in this course.

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 than we can construct from this a verification of A. That is, for arbitrary propositions A:

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

Global completeness follows from local completeness rather directly by induction on the structure of 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 then A is true. Once we add that if 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 area 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.

3 Derived Rules of Inference

One popular device for shortening derivations is to introduce *derived rules of inference*. For example,

$$\frac{A \supset B \ \mathit{true} \quad B \supset C \ \mathit{true}}{A \supset C \ \mathit{true}}$$

is a derived rule of inference. Its derivation is the following:

$$\dfrac{\dfrac{\dfrac{A \supset B \ \mathit{true} \quad \dfrac{}{A \ \mathit{true}}\,u}{B \ \mathit{true}}\,\supset E \qquad B \supset C \ \mathit{true}}{C \ \mathit{true}}\,\supset E}{A \supset C \ \mathit{true}}\,\supset I^u$$

Note that this is simply a hypothetical deduction, using the premises of the derived rule as assumptions. In other words, a derived rule of inference is nothing but an evident hypothetical judgment; its justification is a hypothetical deduction.

We can freely use derived rules in proofs, since any occurrence of such a rule can be expanded by replacing it with its justification.

4 Logical Equivalences

We now consider several classes of logical equivalences in order to develop some intuitions regarding the truth of propositions. Each equivalence has

the form $A \equiv B$, but we consider only the basic connectives and constants ($\wedge$, $\supset$, $\vee$, $\top$, $\bot$) in A and B. Later on we consider negation as a special case. We use some standard conventions that allow us to omit some parentheses while writing propositions. We use the following operator precedences

$$\neg > \wedge > \vee > \supset > \equiv$$

where $\wedge$, $\vee$, and $\supset$ are right associative. For example

$$\neg A \supset A \vee \neg\neg A \supset \bot$$

stands for

$$(\neg A) \supset ((A \vee (\neg(\neg A))) \supset \bot)$$

In ordinary mathematical usage, $A \equiv B \equiv C$ stands for $(A \equiv B) \wedge (B \equiv C)$; in the formal language we do not allow iterated equivalences without explicit parentheses in order to avoid confusion with propositions such as $(A \equiv A) \equiv \top$.

Commutativity. Conjunction and disjunction are clearly commutative, while implication is not.

(C1) $A \wedge B \equiv B \wedge A$ *true*

(C2) $A \vee B \equiv B \vee A$ *true*

(C3) $A \supset B$ is not commutative

Idempotence. Conjunction and disjunction are idempotent, while self-implication reduces to truth.

(I1) $A \wedge A \equiv A$ *true*

(I2) $A \vee A \equiv A$ *true*

(I3) $A \supset A \equiv \top$ *true*

Interaction Laws. These involve two interacting connectives. In principle, there are left and right interaction laws, but because conjunction and disjunction are commutative, some coincide and are not repeated here.

(L1) $A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$ *true*

(L2) $A \wedge \top \equiv A$ *true*

(L3) $A \wedge (B \supset C)$ do not interact

(L4) $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ *true*

(L5) $A \wedge \bot \equiv \bot$ *true*

(L6) $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ *true*

(L7) $A \vee \top \equiv \top$ *true*

(L8) $A \vee (B \supset C)$ do not interact

(L9) $A \vee (B \vee C) \equiv (A \vee B) \vee C$ *true*

(L10) $A \vee \bot \equiv A$ *true*

(L11) $A \supset (B \wedge C) \equiv (A \supset B) \wedge (A \supset C)$ *true*

(L12) $A \supset \top \equiv \top$ *true*

(L13) $A \supset (B \supset C) \equiv (A \wedge B) \supset C$ *true*

(L14) $A \supset (B \vee C)$ do not interact

(L15) $A \supset \bot$ do not interact

(L16) $(A \wedge B) \supset C \equiv A \supset (B \supset C)$ *true*

(L17) $\top \supset C \equiv C$ *true*

(L18) $(A \supset B) \supset C$ do not interact

(L19) $(A \vee B) \supset C \equiv (A \supset C) \wedge (B \supset C)$ *true*

(L20) $\bot \supset C \equiv \top$ *true*

Lecture Notes on Proofs as Programs

15-317: Constructive Logic
Frank Pfenning

Lecture 4
September 3, 2009

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, 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:

$M : A$ $\quad$ 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 term that represents the proof of 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 $M : A$. 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.

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*.

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

The elimination rules correspond to the projections from a pair to its first and second elements.

$$\frac{M : A \wedge B}{\mathbf{fst}\, M : A}\ \wedge E_L \qquad \frac{M : A \wedge B}{\mathbf{snd}\, M : B}\ \wedge E_R$$

Hence conjunction $A \wedge B$ corresponds to the product type $A \times B$.

Truth. Constructively, we think of a proof of $\top$ *true* as a unit element that carries now information.

$$\frac{}{\langle\,\rangle : \top}\ \top I$$

Hence $\top$ corresponds to the unit type **1** with one element. There is no elimination rule and hence no further proof term constructs for truth.

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 $f(x) = \ldots$ where the right-hand side "$\ldots$" 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).

We now use the notation of λ-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.

$$\dfrac{\dfrac{}{u : A}\,u \\ \vdots \\ M : B}{\lambda u{:}A.\ M : A \supset B}\ \supset I^u$$

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.

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

$$\dfrac{\dfrac{}{A\ \mathit{true}}\,u}{A \supset A\ \mathit{true}}\ \supset I^u$$

If we annotate the deduction with proof terms, we obtain

$$\dfrac{\dfrac{}{u : A}\,u}{(\lambda u{:}A.\ u) : A \supset A}\ \supset I^u$$

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

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 $M(N)$.

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

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 $A \to B$. The introduction and elimination rules for implication can also be viewed as formation rules for functional abstraction $\lambda u{:}A.\ M$ and application $M\ N$.

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 *true*.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{B\ \mathit{true}}\wedge E_R \quad \dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{A\ \mathit{true}}\wedge E_L}{B \wedge A\ \mathit{true}}\wedge I}{(A \wedge B) \supset (B \wedge A)\ \mathit{true}}\supset I^u$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \wedge B}\,u}{\mathbf{snd}\, u : B}\wedge E_R \quad \dfrac{\dfrac{}{u : A \wedge B}\,u}{\mathbf{fst}\, u : A}\wedge E_L}{\langle \mathbf{snd}\, u, \mathbf{fst}\, u\rangle : B \wedge A}\wedge I}{(\lambda u.\ \langle \mathbf{snd}\, u, \mathbf{fst}\, u\rangle) : (A \wedge B) \supset (B \wedge A)}\supset I^u$$

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 $A + B$, and the two introduction rules correspond to the left and right injection into a sum type.

$$\dfrac{M : A}{\mathbf{inl}^B\, M : A \vee B}\vee I_L \quad \dfrac{N : B}{\mathbf{inr}^A\, N : A \vee B}\vee I_R$$

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. The elimination rule corresponds to a case construct which discriminates between a left and right injection into a sum types.

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

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 ($\bot$). We can therefore view it as the empty type **0**. The corresponding elimination rule allows a term of $\bot$ 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.

$$\frac{M : \bot}{\mathbf{abort}^C\, M : C}\ \bot E$$

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

This completes our assignment of proof terms to the logical inference rules. Now we can interpret the interaction laws we introduced early as programming exercises. Consider the following distributivity law:

(L11a) $\quad (A \supset (B \wedge C)) \supset (A \supset B) \wedge (A \supset C)$ *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:

$$\lambda u.\ \langle (\lambda w.\ \mathbf{fst}\,(u\,w)), (\lambda v.\ \mathbf{snd}\,(u\,v)) \rangle$$

The following deduction provides the evidence:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \supset (B \wedge C)}\,u \quad \dfrac{}{w : A}\,w}{u\,w : B \wedge C}\supset E}{\mathbf{fst}\,(u\,w) : B}\wedge E_L}{\lambda w.\ \mathbf{fst}\,(u\,w) : A \supset B}\supset I^w \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{u : A \supset (B \wedge C)}\,u \quad \dfrac{}{v : A}\,v}{u\,v : B \wedge C}\supset E}{\mathbf{snd}\,(u\,v) : C}\wedge E_R}{\lambda v.\ \mathbf{snd}\,(u\,v) : A \supset C}\supset I^v}{\langle (\lambda w.\ \mathbf{fst}\,(u\,w)), (\lambda v.\ \mathbf{snd}\,(u\,v)) \rangle : (A \supset B) \wedge (A \supset C)}\wedge I}{\lambda u.\ \langle (\lambda w.\ \mathbf{fst}\,(u\,w)), (\lambda v.\ \mathbf{snd}\,(u\,v)) \rangle : (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))}\supset I^u$$

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.

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* $M \Longrightarrow_R M'$, read "M *reduces to* M'". A computation then proceeds by a sequence of reductions $M \Longrightarrow_R M_1 \Longrightarrow_R M_2 \ldots$, 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.

$$\begin{array}{rcl} \mathbf{fst}\,\langle M, N\rangle & \Longrightarrow_R & M \\ \mathbf{snd}\,\langle M, N\rangle & \Longrightarrow_R & N \end{array}$$

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 λ-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

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

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 λ-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

$$(\lambda u{:}A.\ M)\, N \quad \Longrightarrow_R \quad [N/u]M$$

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.

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

$$\begin{array}{lcl} \textbf{case}\,\textbf{inl}^B\, M\ \textbf{of}\ \textbf{inl}\, u \Rightarrow N \mid \textbf{inr}\, w \Rightarrow O & \Longrightarrow_R & [M/u]N \\ \textbf{case}\,\textbf{inr}^A\, M\ \textbf{of}\ \textbf{inl}\, u \Rightarrow N \mid \textbf{inr}\, w \Rightarrow O & \Longrightarrow_R & [M/w]O \end{array}$$

Falsehood. Since there is no constructor for the empty type there is no reduction rule for falsehood.

This concludes the definition of the reduction judgment. In the next section we will prove some of its properties.

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.

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

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

$$\begin{array}{rcl} \mathsf{comp}\,\langle f, g\rangle\,(w) & = & g(f(w)) \\ \mathsf{comp}\,\langle f, g\rangle & = & \lambda w.\ g(f(w)) \\ \mathsf{comp}\,u & = & \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)(w)) \\ \mathsf{comp} & = & \lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) \end{array}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : (A \supset B) \wedge (B \supset C)}\,u}{\mathbf{snd}\,u : B \supset C}\,\wedge E_R \qquad \dfrac{\dfrac{\dfrac{}{u : (A \supset B) \wedge (B \supset C)}\,u}{\mathbf{fst}\,u : A \supset B}\,\wedge E_L \quad \dfrac{}{w : A}\,w}{(\mathbf{fst}\,u)\,w : B}\,\supset E}{(\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) : C}\,\supset E}{\lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) : A \supset C}\,\supset I^w}{(\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w)) : ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)}\,\supset I^u$$

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

$$(\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w))\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle \quad : \quad A \supset A$$

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

$$\begin{array}{ll} & (\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w))\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle \\ \Longrightarrow_R & \lambda w.\ (\mathbf{snd}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,((\mathbf{fst}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,w) \\ \Longrightarrow_R & \lambda w.\ (\lambda y.\ y)\,((\mathbf{fst}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,w) \\ \Longrightarrow_R & \lambda w.\ (\lambda y.\ y)\,((\lambda x.\ x)\,w) \\ \Longrightarrow_R & \lambda w.\ (\lambda y.\ y)\,w \\ \Longrightarrow_R & \lambda w.\ w \end{array}$$

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 Expansion

We saw in the previous section that proof reductions that witness local soundness form the basis for the computational interpretation of proofs.

Less relevant to computation are the local expansions. 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 $\lambda u.\ M$ for some M.

We can derive what the local expansion must be by annotating the deductions witnessing local expansions from Lecture 3 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 destructor to M and then reconstruct a term of the original type. The resulting rules can be found in Figure 3.

5 Summary of Proof Terms

Judgments.

$M : A$	M is a proof term for proposition A, see Figure 1
$M \Longrightarrow_R M'$	M reduces to M', see Figure 2
$M : A \Longrightarrow_E M'$	M expands to M', see Figure 3

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
$\dfrac{M : A \quad N : B}{\langle M, N\rangle : A \wedge B}\ \wedge I$	$\dfrac{M : A \wedge B}{\mathbf{fst}\, M : A}\ \wedge E_L$ $\dfrac{M : A \wedge B}{\mathbf{snd}\, M : B}\ \wedge E_R$
$\dfrac{}{\langle\,\rangle : \top}\ \top I$	no destructor for $\top$
$\dfrac{\begin{array}{c}\overline{u : A}^{\,u} \\ \vdots \\ M : B\end{array}}{\lambda u{:}A.\ M : A \supset B}\ \supset I^u$	$\dfrac{M : A \supset B \quad N : A}{M\ N : B}\ \supset E$
$\dfrac{M : A}{\mathbf{inl}^B\, M : A \vee B}\ \vee I_L$ $\dfrac{N : B}{\mathbf{inr}^A\, N : A \vee B}\ \vee I_R$	$\dfrac{M : A \vee B \quad \begin{array}{c}\overline{u : A}^{\,u} \\ \vdots \\ N : C\end{array} \quad \begin{array}{c}\overline{w : B}^{\,w} \\ \vdots \\ O : C\end{array}}{\mathbf{case}\, M\ \mathbf{of}\ \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O : C}\ \vee E^{u,w}$
no constructor for $\bot$	$\dfrac{M : \bot}{\mathbf{abort}^C\, M : C}\ \bot E$

Figure 1: Proof term assignment for natural deduction

$$
\begin{array}{rcl}
\textbf{fst}\,\langle M, N\rangle & \Longrightarrow_R & M \\
\textbf{snd}\,\langle M, N\rangle & \Longrightarrow_R & N \\
\multicolumn{3}{c}{\text{no reduction for } \langle\,\rangle} \\
(\lambda u{:}A.\ M)\,N & \Longrightarrow_R & [N/u]M \\
\textbf{case}\,\textbf{inl}^B\,M\ \textbf{of}\ \textbf{inl}\,u \Rightarrow N \mid \textbf{inr}\,w \Rightarrow O & \Longrightarrow_R & [M/u]N \\
\textbf{case}\,\textbf{inr}^A\,M\ \textbf{of}\ \textbf{inl}\,u \Rightarrow N \mid \textbf{inr}\,w \Rightarrow O & \Longrightarrow_R & [M/w]O \\
\multicolumn{3}{c}{\text{no reduction for } \textbf{abort}}
\end{array}
$$

Figure 2: Proof term reductions

$$
\begin{array}{rcl}
M : A \wedge B & \Longrightarrow_E & \langle \textbf{fst}\,M, \textbf{snd}\,M\rangle \\
M : A \supset B & \Longrightarrow_E & \lambda u{:}A.\ M\,u \quad \text{for } u \text{ not free in } M \\
M : \top & \Longrightarrow_E & \langle\,\rangle \\
M : A \vee B & \Longrightarrow_E & \textbf{case}\,M\ \textbf{of}\ \textbf{inl}\,u \Rightarrow \textbf{inl}^B\,u \mid \textbf{inr}\,w \Rightarrow \textbf{inr}^A\,w \\
M : \bot & \Longrightarrow_E & \textbf{abort}^{\bot}\,M
\end{array}
$$

Figure 3: Proof term expansions

Lecture Notes on Quantification

15-317: Constructive Logic
Frank Pfenning

Lecture 5
September 8, 2009

1 Introduction

In this lecture, we introduce universal and existential quantification. 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 τ. Whatever we derive, will of course also hold for specific domain (for example, $\tau = \mathsf{nat}$). The basic judgment connecting objects t to types τ 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.

2 Universal Quantification

First, universal quantification, written as $\forall x{:}\tau.\ A(x)$. Here x is a bound variable and can therefore be renamed as discussed before. When we write $A(x)$ we mean an arbitrary proposition which may depend on x. We will also say that A is *predicate* on elements of type τ.

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.

$$\dfrac{\begin{array}{c}\overline{a : \tau} \\ \vdots \\ A(a)\ \mathit{true}\end{array}}{\forall x{:}\tau.\ A(x)\ \mathit{true}}\ \forall I^a$$

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 $A(a)$ *true*. In particular, it may not occur in $\forall x{:}\tau.\ A(x)$.

If we think of this 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 $A(t)$ *true*.

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

We must verify that $t : \tau$ so that $A(t)$ is a well-formed proposition.

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

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

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 local reduction for universal quantification then exploits this substitution principle.

$$\dfrac{\dfrac{\begin{array}{c}\overline{a : \tau} \\ \mathcal{D} \\ A(a)\ \mathit{true}\end{array}}{\forall x{:}\tau.\ A(x)\ \mathit{true}}\ \forall I^a \qquad \begin{array}{c}\mathcal{E} \\ t : \tau\end{array}}{A(t)\ \mathit{true}}\ \forall E \qquad \Longrightarrow_R \qquad \begin{array}{c}\dfrac{\mathcal{E}}{t : \tau} \\ [t/a]\mathcal{D} \\ A(t)\ \mathit{true}\end{array}$$

The local expansion introduces a parameter which we can use to elimi-

nate the universal quantifier.

$$\dfrac{\overset{\mathcal{D}}{\forall x{:}\tau.\ A(x)\ \mathit{true}}}{} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\overset{\mathcal{D}}{\forall x{:}\tau.\ A(x)\ \mathit{true}} \quad \overline{a : \tau}}{A(a)\ \mathit{true}}\ \forall E}{\forall x{:}\tau.\ A(x)\ \mathit{true}}\ \forall I^a$$

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

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

3 Existential Quantification

The existential quantifier is more difficult to specify, although the introduction rule seems innocuous enough.

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

The elimination rules creates some difficulties. We cannot write

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

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'.

The best we can do is to assume that $A(a)$ is true for some new parameter a. The scope of this assumption is limited to the proof of some

conclusion C *true* which does not mention a (which must be new).

$$\dfrac{\exists x{:}\tau.\ A(x)\ \mathit{true} \qquad \begin{array}{c}\overline{a:\tau} \quad \overline{A(a)\ \mathit{true}}\ u \\ \vdots \\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\ \exists E^{a,u}$$

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. We use this crucially in the local reduction.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D} \\ t:\tau\end{array} \quad \begin{array}{c}\mathcal{E} \\ A(t)\ \mathit{true}\end{array}}{\exists x{:}\tau.\ A(x)\ \mathit{true}}\ \exists I \qquad \begin{array}{c}\overline{a:\tau} \quad \overline{A(a)\ \mathit{true}}\ u \\ \mathcal{F} \\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\ \exists E^{a,u} \qquad \Longrightarrow_R \qquad \dfrac{\begin{array}{c}\mathcal{D} \\ t:\tau\end{array} \quad \dfrac{\begin{array}{c}\mathcal{E} \\ A(t)\ \mathit{true}\end{array}}{[t/a]\mathcal{F}}\ u}{C\ \mathit{true}}$$

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

The local expansion is patterned after the disjunction.

$$\begin{array}{c}\mathcal{D} \\ \exists x{:}\tau.\ A(x)\ \mathit{true}\end{array} \qquad \Longrightarrow_E \qquad \dfrac{\begin{array}{c}\mathcal{D} \\ \exists x{:}\tau.\ A(x)\ \mathit{true}\end{array} \quad \dfrac{\overline{a:\tau} \quad \overline{A(a)\ \mathit{true}}\ u}{\exists x{:}\tau.\ A(x)\ \mathit{true}}\ \exists I}{\exists x{:}\tau.\ A(x)\ \mathit{true}}\ \exists E^{a,u}$$

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 possibly dependence on a bound variable is indicated by writing a general *predicate* $A(x_1, \ldots, x_n)$. 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.

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

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\exists x{:}\tau.\ A(x)) \supset C\ \mathit{true}}\,u \quad \dfrac{}{a:\tau} \quad \dfrac{}{A(a)\ \mathit{true}}\,w}{\vdots}}{C\ \mathit{true}}}{A(a) \supset C\ \mathit{true}}\,\supset I^w}{\forall x{:}\tau.\ A(x) \supset C\ \mathit{true}}\,\forall I^a}{((\exists x{:}\tau.\ A(x)) \supset C) \supset \forall x{:}\tau.\ (A(x) \supset C)\ \mathit{true}}\,\supset I^u$$

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 $A(a)$ *true* are atomic. This gives us a new subgoal.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\exists x{:}\tau.\ A(x)) \supset C\ \mathit{true}}\,u \qquad \dfrac{\dfrac{}{a:\tau} \quad \dfrac{}{A(a)\ \mathit{true}}\,w}{\vdots}}{\exists x{:}\tau.\ A(x)}\,\supset E}{C\ \mathit{true}}}{A(a) \supset C\ \mathit{true}}\,\supset I^w}{\forall x{:}\tau.\ A(x) \supset C\ \mathit{true}}\,\forall I^a}{((\exists x{:}\tau.\ A(x)) \supset C) \supset \forall x{:}\tau.\ (A(x) \supset C)\ \mathit{true}}\,\supset I^u$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\exists x{:}\tau.\ A(x)) \supset C\ \mathit{true}}\,u \quad \dfrac{\dfrac{}{a:\tau} \quad \dfrac{}{A(a)\ \mathit{true}}\,w}{\exists x{:}\tau.\ A(x)}\,\exists I}{C\ \mathit{true}}\,\supset E}{A(a) \supset C\ \mathit{true}}\,\supset I^w}{\forall x{:}\tau.\ A(x) \supset C\ \mathit{true}}\,\forall I^a}{((\exists x{:}\tau.\ A(x)) \supset C) \supset \forall x{:}\tau.\ (A(x) \supset C)\ \mathit{true}}\,\supset I^u$$

We now consider the reverse implication.

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x{:}\tau.\ A(x)\ \mathit{true}}\ w \quad \dfrac{}{\forall x{:}\tau.\ A(x) \supset C\ \mathit{true}}\ u}{\vdots}}{C\ \mathit{true}}}{(\exists x{:}\tau.\ A(x)) \supset C\ \mathit{true}}\ \supset I^w}{(\forall x{:}\tau.\ (A(x) \supset C)) \supset ((\exists x{:}\tau.\ A(x)) \supset C)\ \mathit{true}}\ \supset I^u$$

No 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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x{:}\tau.\ A(x)\ \mathit{true}}\ w \qquad \dfrac{\dfrac{}{\forall x{:}\tau.\ A(x) \supset C\ \mathit{true}}\ u \quad \dfrac{}{a : \tau} \quad \dfrac{}{A(a)\ \mathit{true}}\ v}{\vdots}}{C\ \mathit{true}}\ \exists E^{a,v}}{C\ \mathit{true}}}{(\exists x{:}\tau.\ A(x)) \supset C\ \mathit{true}}\ \supset I^w}{(\forall x{:}\tau.\ (A(x) \supset C)) \supset ((\exists x{:}\tau.\ A(x)) \supset C)\ \mathit{true}}\ \supset I^u$$

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x{:}\tau.\ A(x)\ \mathit{true}}\ w \qquad \dfrac{\dfrac{\dfrac{}{\forall x{:}\tau.\ A(x) \supset C\ \mathit{true}}\ u \quad \dfrac{}{a : \tau}}{A(a) \supset C\ \mathit{true}}\ \forall E \quad \dfrac{}{A(a)\ \mathit{true}}\ v}{\vdots}}{C\ \mathit{true}}\ \exists E^{a,v}}{C\ \mathit{true}}}{(\exists x{:}\tau.\ A(x)) \supset C\ \mathit{true}}\ \supset I^w}{(\forall x{:}\tau.\ (A(x) \supset C)) \supset ((\exists x{:}\tau.\ A(x)) \supset C)\ \mathit{true}}\ \supset I^u$$

Now we can fill the gap with an implication elimination.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x{:}\tau.\ A(x)\ \mathit{true}}\,w \qquad \dfrac{\dfrac{\dfrac{}{\forall x{:}\tau.\ A(x) \supset C\ \mathit{true}}\,u \quad \dfrac{}{a : \tau}}{A(a) \supset C\ \mathit{true}}\,\forall E \quad \dfrac{}{A(a)\ \mathit{true}}\,v}{C\ \mathit{true}}\,\supset E}{C\ \mathit{true}}\,\exists E^{a,v}}{(\exists x{:}\tau.\ A(x)) \supset C\ \mathit{true}}\,\supset I^{w}}{(\forall x{:}\tau.\ (A(x) \supset C)) \supset ((\exists x{:}\tau.\ A(x)) \supset C)\ \mathit{true}}\,\supset I^{u}$$

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:

$$\dfrac{\begin{array}{c}\overline{a : \tau}\\ \vdots \\ A(a)\uparrow\end{array}}{\forall x{:}\tau.\ A(x)\uparrow}\,\forall I^{a} \qquad \dfrac{\forall x{:}\tau.\ A(x)\downarrow \quad t : \tau}{A(t)\downarrow}\,\forall E$$

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.

$$\dfrac{t : \tau \quad A(t)\uparrow}{\exists x{:}\tau.\ A(x)\uparrow}\,\exists I \qquad \dfrac{\exists x{:}\tau.\ A(x)\downarrow \qquad \begin{array}{c}\overline{a : \tau} \quad \overline{A(a)\downarrow}\,u\\ \vdots \\ C\uparrow\end{array}}{C\uparrow}\,\exists E^{a,u}$$

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

$$\dfrac{\dfrac{\dfrac{}{\exists x{:}\tau.\ A(x)\downarrow}\,u \quad \dfrac{}{a:\tau}}{\vdots}}{\dfrac{\dfrac{A(a)\uparrow}{\forall x{:}\tau.\ A(x)\uparrow}\,\forall I^a}{(\exists x{:}\tau.\ A(x)) \supset (\forall x{:}\tau.\ A(x))\uparrow}\,\supset I^u}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x{:}\tau.\ A(x)\downarrow}\,u \qquad \dfrac{\dfrac{}{b:\tau} \quad \dfrac{\dfrac{}{A(b)\downarrow}\,v \quad \dfrac{}{a:\tau}}{\vdots}}{A(a)\uparrow}}{A(a)\uparrow}\,\exists E^{b,v}}{\forall x{:}\tau.\ A(x)\uparrow}\,\forall I^a}{(\exists x{:}\tau.\ A(x)) \supset (\forall x{:}\tau.\ A(x))\uparrow}\,\supset I^u$$

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 $A(a)$.

Lecture Notes on Natural Numbers

15-317: Constructive Logic
Frank Pfenning

Lecture 6
September 10, 2009

1 Introduction

In this lecture we discuss the 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 as introduced in the 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*.

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 $t : \tau$ 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.

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

$$\frac{}{\mathsf{0} : \mathsf{nat}}\ \mathsf{nat}I_0 \qquad \frac{n : \mathsf{nat}}{\mathsf{s}\,n : \mathsf{nat}}\ \mathsf{nat}I_s$$

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 $\mathsf{nat}I_s$ rules are the same, namely nat. Fortunately, the *term* n in the premise is a part of the term $\mathsf{s}\,n$ in the conclusion, so the definition is not circular.

But what should the elimination rule be? We cannot decompose the proposition into its parts, so we decompose the term instead. This is similar to disjunction in that it proceeds by cases, accounting for the possibility that a given n of type nat is either $\mathsf{0}$ or $\mathsf{s}\,x$ for some x.

$$\frac{n : \mathsf{nat} \quad C(\mathsf{0})\ \textit{true} \qquad \begin{array}{c}\overline{x : \mathsf{nat}} \quad \overline{C(x)\ \textit{true}}^{\,u} \\ \vdots \\ C(\mathsf{s}\,x)\ \textit{true}\end{array}}{C(n)\ \textit{true}}\ \mathsf{nat}E^{x,u}$$

In words: In order to prove property C of a natural number n we have to prove $C(\mathsf{0})$ and also $C(\mathsf{s}\,x)$ 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 $C(\mathsf{0})$ is the base case, and the proof of $C(\mathsf{s}\,x)$ from the assumption $C(x)$ is the induction step.

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:

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

for an arbitrary property C of natural numbers. It is an easy exercise to prove this with the induction rule above.

To illustrate this rule in action, we start with a very simple property: every natural number is either $\mathsf{0}$ or has a predecessor. First, a detailed induction proof in the usual mathematical style.

Theorem: $\forall x{:}\mathsf{nat}.\ x = \mathsf{0} \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y$.
Proof: By induction on x.

Case: $x = \mathsf{0}$. Then the left disjunct is true.

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

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 refl expressing reflexivity of equality on natural numbers ($n = n$).

$$\dfrac{\dfrac{}{x : \mathsf{nat}} \quad \dfrac{\dfrac{\dfrac{}{0 = 0}\,\mathsf{refl}}{0 = 0 \vee \exists y{:}\mathsf{nat}.\ 0 = \mathsf{s}\,y}\,\vee I_L \quad \dfrac{\dfrac{\dfrac{}{x' : \mathsf{nat}} \quad \dfrac{}{\mathsf{s}\,x' = \mathsf{s}\,x'}\,\mathsf{refl}}{\exists y{:}\mathsf{nat}.\ \mathsf{s}\,x' = \mathsf{s}\,y}\,\exists I}{\mathsf{s}\,x' = 0 \vee \exists y{:}\mathsf{nat}.\ \mathsf{s}\,x' = \mathsf{s}\,y}\,\vee I_R}{x = 0 \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y}\,\mathsf{nat}E^{x',u}}{\forall x{:}\mathsf{nat}.\ x = 0 \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y}\,\forall I^x$$

This is a simple proof by cases and does not use the induction hypothesis, which would have been labeled u.

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

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

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

As a second example we specify a function which tests if its argument is even or odd. For this purpose we assume a doubling function $2\times$. Equality is decided by an oracle.

Theorem: $\forall x{:}\mathsf{nat}.\ (\exists y.\ x = 2 \times y) \vee (\exists z.\ x = \mathsf{s}\,(2 \times z))$.
Proof: By induction on x.

Case: $x = 0$. Then pick $y = 0$ since $0 = 2 \times 0$.

Case: $x = \mathsf{s}\,x'$. By induction hypothesis we have either $\exists y.\ x' = 2 \times y$ or $\exists z.\ x' = \mathsf{s}\,(2 \times z)$. We distinguish these two cases.

Case: $\exists y.\ x' = 2 \times y$. Then the second disjunct holds because we can pick $z = y$: $x = \mathsf{s}\,x' = \mathsf{s}\,(2 \times y)$.

Case: $\exists z.\ x' = \mathsf{s}\,(2 \times z)$. Then the first disjunct holds because we can pick $y = \mathsf{s}\,z$: $x = \mathsf{s}\,x' = \mathsf{s}\,(\mathsf{s}\,(2 \times z)) = 2 \times (\mathsf{s}\,z)$ by properties of $2\times$.

We start the transcription of this proof.

$$\dfrac{\dfrac{\dfrac{0 = 2 \times 0 \vee \ldots}{(\exists y.\, 0 = 2 \times y) \vee \ldots}\,\exists I \qquad \begin{array}{c}\overline{x' : \mathsf{nat}} \quad \overline{(\exists y.\, x' = 2 \times y) \vee (\exists z.\, x' = \mathsf{s}\,(2 \times z))}\,u \\ \vdots \\ (\exists y.\, \mathsf{s}\,x' = 2 \times y) \vee (\exists z.\, \mathsf{s}\,x' = \mathsf{s}\,(2 \times z))\end{array}}{(\exists y.\, x = 2 \times y) \vee (\exists z.\, x = \mathsf{s}\,(2 \times z))}\,\mathsf{nat}E^{x',u}}{\forall x{:}\mathsf{nat}.\, (\exists y.\, x = 2 \times y) \vee (\exists z.\, x = \mathsf{s}\,(2 \times z))}\,\forall I^{x}$$

From here, we proceed by an $\vee E$ applied to u, followed by an $\exists E$ in each branch, naming the y and z that are known to exist. Unfortunately, the 2-dimensional notation for natural deductions which is nice and direct for describing and reasoning about the rules, is not so good for writing actual formal deductions.

3 Local Proof Reduction

We check that the rules are locally sound and complete. For soundness, we verify that no matter how we introduce the judgment $n : \mathsf{nat}$, we can find a more direct proof of the conclusion. In the case of $\mathsf{nat}I_0$ this is easy to see, because the second premise already establishes our conclusion.

$$\dfrac{\dfrac{}{0 : nat}\,\mathsf{nat}I_0 \qquad \begin{array}{c}\mathcal{E} \\ C(0)\ true\end{array} \qquad \begin{array}{c}\overline{x : nat} \quad \overline{C(x)\ true}\,u \\ \mathcal{F} \\ C(\mathsf{s}\,x)\ true\end{array}}{C(0)\ true}\,\mathsf{nat}E^{x,u} \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{E} \\ C(0)\ true\end{array}$$

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

$$\dfrac{\dfrac{\mathcal{D}}{\dfrac{n' : \mathsf{nat}}{\mathsf{s}\,n' : \mathsf{nat}}\,\mathsf{nat}I_s} \quad \begin{array}{c}\mathcal{E}\\ C(0)\ \mathit{true}\end{array} \quad \begin{array}{c}\overline{x : \mathsf{nat}} \quad \overline{C(x)\ \mathit{true}}^{\,u}\\ \mathcal{F}\\ C(\mathsf{s}\,x)\ \mathit{true}\end{array}}{C(\mathsf{s}\,n')\ \mathit{true}}\,\mathsf{nat}E^{x,u}$$

$$\Longrightarrow_R \quad \dfrac{\mathcal{D}}{n' : \mathsf{nat}} \quad \dfrac{\begin{array}{ccc}\mathcal{D} & \mathcal{E} & \overline{x : \mathsf{nat}} \quad \overline{C(x)\ \mathit{true}}^{\,u}\\ n' : nat & C(0)\ \mathit{true} & \mathcal{F}\\ & & C(s\,x)\ \mathit{true}\end{array}}{C(n')\ \mathit{true}}\,\mathsf{nat}E^{x,u}$$

$$[n'/x]\mathcal{F}'$$
$$C(s\,n')\ \mathit{true}$$

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 $\mathsf{s}\,n'$ 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.

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 $n : \mathsf{nat}$. The next section will give us the tool.

4 Primitive Recursion

Reconsidering the elimination rule for natural numbers, we can notice that we exploit the knowledge that $n : \mathsf{nat}$, 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 $n : \mathsf{nat}$ when we are trying to establish a judgment of the

form $t : \tau$. The rule then becomes

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

Here, R is a new term constructor,[1] t_0 is the case where $n = 0$, and t_s captures the case where $n = \mathsf{s}\, n'$. In the latter case x is a new parameter introduced in the rule that stands for n'. r stands for the result of the function R when applied to n', which corresponds to an appeal to the induction hypothesis.

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

$$\begin{array}{lcl} R(\mathsf{0}, t_0, x.\, r.\, t_s) & \Longrightarrow_R & t_0 \\ R(\mathsf{s}\, n', t_0, x.\, r.\, t_s) & \Longrightarrow_R & [R(n', t_0, x.\, r.\, t_s)/r][n'/x]\, t_s \end{array}$$

These are still quite unwieldy, so we consider a more readable schematic form, called the *schema of primitive recursion*. If we write

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

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

$$f = \lambda x.\, R(x, t_0, x.\, r.\, t_s(x, r).$$

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

$$\begin{array}{rcl} f(\mathsf{0}) & \Longrightarrow_R & R(\mathsf{0}, t_0, x.\, r.\, t_s(x, r)) \\ & \Longrightarrow_R & t_0 \end{array}$$

and

$$\begin{array}{rcl} f(\mathsf{s}\, n) & \Longrightarrow_R & R(\mathsf{s}\, n, t_0, x.\, r.\, t_s(x, r)) \\ & \Longrightarrow_R & t_s(n, R(n, t_0, x.\, r.\, t_s(x, r))) \\ & = & t_s(n, f(n)) \end{array}$$

[1] R suggests recursion

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

To be completely formal, we would also have to define the function space on data, which comes from the following pair of introduction and elimination rules for $\tau \to \sigma$. Since they are completely analogous to implication, except for using terms instead of proof terms, we will not discuss them further

$$\frac{\begin{array}{c}\overline{x : \tau} \\ \vdots \\ s : \sigma\end{array}}{\lambda x{:}\tau.\ s : \tau \to \sigma}\ {\to}I \qquad \frac{s : \tau \to \sigma \quad t : \tau}{s\,t : \sigma}\ {\to}E$$

The local reduction is

$$(\lambda x{:}\tau.\ s)\,t \quad \Longrightarrow_R \quad [t/x]s$$

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

$$\begin{array}{lcl} \mathsf{double}(\mathsf{0}) & = & \mathsf{0} \\ \mathsf{double}(\mathsf{s}\,x) & = & \mathsf{s}\,(\mathsf{s}\,(\mathsf{double}\,x)) \end{array}$$

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

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

From now on we will be content with using the schema of primitive recursion. We define addition and multiplication, as exercises.

$$\begin{array}{lcl} \mathsf{plus}(\mathsf{0}) & = & \lambda y.\ y \\ \mathsf{plus}(\mathsf{s}\,x) & = & \lambda y.\ \mathsf{s}\,((\mathsf{plus}\,x)\,y) \end{array}$$

$$\begin{array}{lcl} \mathsf{times}(\mathsf{0}) & = & \lambda y.\ \mathsf{0} \\ \mathsf{times}(\mathsf{s}\,x) & = & \lambda y.\ (\mathsf{plus}\ ((\mathsf{times}\,x)\,y))\,y \end{array}$$

5 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.

$$\dfrac{n : \mathsf{nat} \quad M_0 : C(\mathsf{0})\ \mathit{true} \qquad \begin{array}{c} \overline{x : \mathsf{nat}} \quad \overline{u : C(x)\ \mathit{true}}^{\,u} \\ \vdots \\ M_s : C(\mathsf{s}\,x)\ \mathit{true} \end{array}}{R(n, M_0, x.\,u.\,M_s) : C(n)\ \mathit{true}}\ \mathsf{nat}E^{x,u}$$

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

$$\begin{array}{lcl} R(\mathsf{0}, M_0, x.\,u.\,M_s) & \Longrightarrow_R & M_0 \\ R(\mathsf{s}\,n', M_0, x.\,u.\,M_s) & \Longrightarrow_R & [R(n', M_0, x.\,u.\,M_s)/u][n'/x]\,M_s \end{array}$$

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: $\forall x{:}\mathsf{nat}.\ x = \mathsf{0} \vee \exists y{:}\mathsf{nat}.\ x = \mathsf{s}\,y$.
Proof: By induction on x.

Case: $x = \mathsf{0}$. Then the left disjunct is true.

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

The extracted function has the form

$$\mathsf{pred} \quad = \quad \lambda x{:}\mathsf{nat}.\ R(x, \mathbf{inl}\ \_, x.\,r.\,\mathbf{inr}\langle x, \_\rangle)$$

More easily readable is the ML version, where we have eliminated the computationally irrelevant parts from above.

```
datatype nat = Z | S of nat;
datatype nat_option = Inl | Inr of nat
(* pred : nat -> nat_option *)
fun pred Z = Inl
  | pred (S x) = Inr x
```

Theorem: $\forall x{:}\mathsf{nat}.\ (\exists y.\ x = 2 \times y) \vee (\exists z.\ x = \mathsf{s}\,(2 \times z))$.
Proof: By induction on x.

Case: $x = 0$. Then pick $y = 0$ since $0 = 2 \times 0$.

Case: $x = \mathsf{s}\,x'$. By induction hypothesis we have either $\exists y.\ x' = 2 \times y$ or $\exists z.\ x' = \mathsf{s}\,(2 \times z)$. We distinguish these two cases.

Case: $\exists y.\ x' = 2 \times y$. Then the second disjunct holds because we can pick $z = y$: $x = \mathsf{s}\,x' = \mathsf{s}\,(2 \times y)$.

Case: $\exists z.\ x' = \mathsf{s}\,(2 \times z)$. Then the first disjunct holds because we can pick $y = \mathsf{s}\,z$: $x = \mathsf{s}\,x' = \mathsf{s}\,(\mathsf{s}\,(2 \times z)) = 2 \times (\mathsf{s}\,z)$ by properties of $2 \times$.

$$
\begin{array}{rcl}
\mathsf{half} & = & \lambda x{:}\mathsf{nat}.\ R(x, \mathbf{inl}\langle 0, \_\rangle, \\
& & \quad x.\,r.\,\mathbf{case}\ r\ \ \mathbf{of}\ \ \mathbf{inl}\,u \Rightarrow \mathbf{let}\ \langle y, \_\rangle = u\ \ \mathbf{in}\ \ \mathbf{inr}\,\langle y, \_\rangle \\
& & \qquad\qquad \mid \mathbf{inr}\,w \Rightarrow \mathbf{let}\ \langle z, \_\rangle = w\ \ \mathbf{in}\ \ \mathbf{inl}\,\langle \mathsf{s}\,z, \_\rangle
\end{array}
$$

or, in ML, where `half(`$2 \times n$`)` returns `Even(`n`)` and `half(`$2 \times n + 1$`)` returns `Odd(`n`)`.

```
datatype nat = Z | S of nat;
datatype parity = Even of nat | Odd of nat
(* half : nat -> parity *)
fun half Z = Even Z
  | half (S x) = (case half x
                    of Even y => Odd y
                     | Odd z => Even (S z))
```

6 Local Expansion

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

$$
\begin{array}{c}\mathcal{D} \\ n : \mathsf{nat}\end{array}
\quad \Longrightarrow_E \quad
\dfrac{\begin{array}{ccc}\mathcal{D} & & \dfrac{}{x : \mathsf{nat}} \\ n : \mathsf{nat} & \dfrac{}{0 : \mathsf{nat}}\ \mathsf{nat}I_0 & \dfrac{}{\mathsf{s}\,x : \mathsf{nat}}\ \mathsf{nat}I_s \end{array}}{n : \mathsf{nat}}\ \mathsf{nat}E^{x,r}
$$

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. This reflects that simple proof-by-cases would also be 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.

7 Equality

With primitive recursion we have sufficient expressive power to define functions like double, and also addition, multiplication, exponentiation, and many others. But what about predicates such as equality or inequality? Fortunately, our method of using introduction and elimination rules works as before.

$$\frac{}{\mathsf{0} = \mathsf{0}\ \mathit{true}}\ {=}I_{00} \qquad \frac{n = n'\ \mathit{true}}{\mathsf{s}\,n = \mathsf{s}\,n'\ \mathit{true}}\ {=}I_{ss}$$

If we take this as our definition of equality on natural numbers, how can we use the knowledge that $n = n'$? If n and n' 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 and we can derive anything.

no rule ${=}E_{00} \qquad \frac{\mathsf{0} = \mathsf{s}\,n\ \mathit{true}}{C\ \mathit{true}}\ {=}E_{0s} \qquad \frac{\mathsf{s}\,n = \mathsf{0}\ \mathit{true}}{C\ \mathit{true}}\ {=}E_{s0} \qquad \frac{\mathsf{s}\,n = \mathsf{s}\,n'\ \mathit{true}}{n = n'\ \mathit{true}}\ {=}E_{ss}$

Local soundness is very easy to check, but what about local completeness? It turns out we cannot really write it without implication, which is further testimony to some inherent incompleteness in arithmetic.

Lecture Notes on Classical Logic

15-317: Constructive Logic
William Lovas

Lecture 7
September 15, 2009

1 Introduction

In this lecture, we design a judgmental formulation of classical logic. To gain an intuition, we explore various equivalent notions of the essence of classical reasoning including the Law of the Excluded Middle and Double-Negation Elimination. Throughout the discussion a common theme is the indirectness and "dishonesty" of classical proofs, an idea which will later be key to understanding their computational interpretation. Eventually, we arrive at a judgmentally parsimonious system based on the principle of Proof by Contradiction and founded on two new forms of judgment: *A is false* (written *A false*) and $contradiction$ (written #).

2 Example

Classical reasoning is pervasive in classical mathematics, so we begin with a typical example of a theorem proven using classical methods.

> **Theorem:** $\exists a, b \in \mathbb{R}$. *irrational*$(a) \wedge$ *irrational*$(b) \wedge$ *rational*(a^b)
>
> **Proof:** Consider $\sqrt{2}^{\sqrt{2}}$: this number is either rational or irrational. Suppose it is rational: then $a = \sqrt{2}$, $b = \sqrt{2}$ gives the required result. Suppose it is not: then $a = \sqrt{2}^{\sqrt{2}}$, $b = \sqrt{2}$ gives the required result, as $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = (\sqrt{2})^2 = 2$.

Although this claims to be a proof of an existential theorem, it gives the reader no grasp on what numbers actually witness the result—it offers two possible choices but no further guidance as to which one is correct.

A slightly whimsical characterization of the proof offers a glimpse into the computational content of classical reasoning. Imagine a prominent mathematician delivering the above proof aloud as part of a lecture before a large audience. Initially, he delivers only the first half, saying, "Let $a = \sqrt{2}$ and $b = \sqrt{2}$; then a and b are irrational while a^b is rational—*trust me*." Amidst some mumbling, the audience nods and accepts the proof; after all, he *is* a very prominent mathematician, and he probably knows what he's talking about. But then, halfway through the lecture, a student from the back suddenly leaps up and exclaims, "His proof is no good—*I* have a proof that $\sqrt{2}^{\sqrt{2}}$ is *irrational!*"[1]. The audience gasps and a murmur runs through the crowd, but before anyone else can speak, the mathematician calmly responds, "May I see your proof?" After checking it over, the mathematician addresses the crowd again: "My apologies—I misspoke earlier. What I meant to say was this: Let $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$; then a and b are irrational—I have this proof, if you don't believe me!—while $a^b = 2$ and is therefore rational."

The poor student at the back thought she could attain fame and fortune by debunking the prominent mathematician, but in fact, the mathematician stole the glory by leveraging her proof for his own ends. Classical proofs exhibit a similar time-traveling behavior when executed, as we'll see in the next lecture.

3 What is classical logic?

Classical logic can be characterized by a number of equivalent axioms:

- Proof by Contradiction
- Law of the Excluded Middle: $A \vee \neg A$
- Double-Negation Elimination: $\neg\neg A \supset A$
- Peirce's Law: $((A \supset B) \supset A) \supset A$

We might consider making the logic we've seen so far classical by adding one or more rules that correspond to these axioms. For instance, we might

[1] It is, in fact, though the proof is non-trivial.

add:

$$\frac{}{A \vee \neg A \ \mathit{true}}\ \mathit{LEM} \qquad \text{or} \qquad \frac{\neg\neg A \ \mathit{true}}{A \ \mathit{true}}\ \mathit{DNE}.$$

Of course, we need only add one or the other, and not both, since they are interderivable. First, let's show how we can derive DNE from LEM:

$$\dfrac{\dfrac{}{A \vee \neg A \ \mathit{true}}\ \mathit{LEM} \quad \dfrac{}{A \ \mathit{true}}\ u \qquad \dfrac{\neg\neg A \ \mathit{true} \quad \dfrac{}{\neg A \ \mathit{true}}\ v}{\dfrac{\bot \ \mathit{true}}{A \ \mathit{true}}\ \bot E}\ \supset E}{A \ \mathit{true}}\ \vee E^{u,v}$$

As you can see, it is precisely the power of having the middle excluded that lets us turn our proof of $\neg\neg A \ \mathit{true}$ into a proof of $\bot \ \mathit{true}$, and thus a proof of $A \ \mathit{true}$ as required.

Using DNE also allows us to derive LEM. Note that since the LEM rule has no premises, this will have to be a completely closed derivation. To help elucidate the thought process of a classical prover, we'll do a step-by-step derivation. We start bottom up from the conclusion:

$$\begin{array}{c}\vdots\\ A \vee \neg A \ \mathit{true}\end{array}$$

If we were to attempt to proceed as we've done previously, we would now have to apply a disjunction-introduction rule, but we have no way of deciding which injection to choose: merely positing all propositions to be true or not does nothing to make apparent which is the case. We know that this classical tautology is unprovable intuitionistically, anyhow, so we have no hope but to begin by employing our classical rule, double-negation elimination.

$$\dfrac{\begin{array}{c}\vdots\\ \neg\neg(A \vee \neg A) \ \mathit{true}\end{array}}{A \vee \neg A \ \mathit{true}}\ \mathit{DNE}$$

Since $\neg A$ is notationally defined to mean $A \supset \bot$, we can take the usual no-brainer step of applying implication introduction:

$$\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)\ \mathit{true}}\,u}{\vdots}}{\dfrac{\dfrac{\bot\ \mathit{true}}{\neg\neg(A \vee \neg A)\ \mathit{true}}\,\supset I^u}{A \vee \neg A\ \mathit{true}}\,\mathit{DNE}}$$

Now we are tasked with proving falsity true: a difficult task, to be sure, but a glimmer of hope shines through in our sole assumption u. Perhaps by applying it to an appropriate argument, we can conclude $\bot\ \mathit{true}$ as required:

$$\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)\ \mathit{true}}\,u}{\vdots}}{\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)\ \mathit{true}}\,u \qquad A \vee \neg A\ \mathit{true}}{\bot\ \mathit{true}}\,\supset E}{\neg\neg(A \vee \neg A)\ \mathit{true}}\,\supset I^u}{A \vee \neg A\ \mathit{true}}\,\mathit{DNE}$$

At this point, if we were not being careful, we might throw up our hands and quit. We're right back to where we started, trying to prove $A \vee \neg A\ \mathit{true}$! But now we have an extra hypothesis to help us. Undaunted, we pause to consider: which shall we prove, A or $\neg A$? Since we know nothing of the structure of A, we have no hope of proving it unless an assumption can yield it, but nothing seems appropriate. So instead, we try for the right disjunct, $\neg A$. We can also go ahead and apply implication introduction without thinking twice, this time attaining an assumption v that $A\ \mathit{true}$.

$$\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)\ \mathit{true}}\,u,\ \dfrac{}{A\ \mathit{true}}\,v}{\vdots}}{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)\ \mathit{true}}\,u \qquad \dfrac{\dfrac{\bot\ \mathit{true}}{\neg A\ \mathit{true}}\,\supset I^v}{A \vee \neg A\ \mathit{true}}\,\vee I_R}{\bot\ \mathit{true}}\,\supset E}{\neg\neg(A \vee \neg A)\ \mathit{true}}\,\supset I^u}{A \vee \neg A\ \mathit{true}}\,\mathit{DNE}$$

This situation is familiar once again: to prove $\bot$ *true*, we must use an assumption, and u still seems to be the only one that can help. So we carry out another implication elimination, feeling only the slightest sense of *deja vu*...

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)\ \mathit{true}}\,u}{\vdots}}{A \vee \neg A\ \mathit{true}}\,\supset E}{\bot\ \mathit{true}}\,\supset I^{v}}{\neg A\ \mathit{true}}\,\vee I_R}{A \vee \neg A\ \mathit{true}}\,\supset E \qquad \dfrac{}{A\ \mathit{true}}\,v$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)\ \mathit{true}}\,u}{\bot\ \mathit{true}}\,\supset I^{u}}{\neg\neg(A \vee \neg A)\ \mathit{true}}\,\mathit{DNE}}{A \vee \neg A\ \mathit{true}}$$

Once more we are faced with our great adversary, the *tertium non datur* himself! But now the tables are turned: no longer must we cower in fear behind our security blanket of negation. No, now we can prove A *true* itself, by the very assumption we hypothesized to prove $\neg A$ *true*! The derivation is now complete.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)\ \mathit{true}}\,u \qquad \dfrac{\dfrac{}{A\ \mathit{true}}\,v}{A \vee \neg A\ \mathit{true}}\,\vee I_L}{\bot\ \mathit{true}}\,\supset E}{\neg A\ \mathit{true}}\,\supset I^{v}}{A \vee \neg A\ \mathit{true}}\,\vee I_R}{\bot\ \mathit{true}}\,\supset E}{\neg\neg(A \vee \neg A)\ \mathit{true}}\,\supset I^{u}}{A \vee \neg A\ \mathit{true}}\,\mathit{DNE}$$

An interesting point to note about this proof is that, save for the last line, it is a perfectly valid intuitionistic proof. Only at the very end—or beginning, as we've told the story—did we need to appeal to classical reasoning in the form of *DNE*. In fact, this observation hints at a more general one: there is a double-negation translation, discussed in the next lecture, that translates any classical theorem into one that is intuitionistic valid.

From the intuitionistic portion of the deduction, we can read off a proof term, the analysis of which will begin to reveal the nature of the computational interpretation of classical proofs.

$$\mathbf{fn}\ (u : \neg(A \vee \neg A)) \Rightarrow u\ (\mathbf{inr}\ (\mathbf{fn}\ (v : A) \Rightarrow u\ (\mathbf{inl}\ v)))$$

It begins by supposing a proof u of $\neg(A \vee \neg A)$—i.e., a *refutation* of $A \vee \neg A$—and then proceeds to debunk that refutation, showing that it must have been mistaken by driving it to a contradiction. There are two choices of how to do so: prove A or prove $\neg A$. First, it chooses to show that, in fact, $\neg A$ is the case. Its proof of $\neg A$ proceeds as usual, supposing a proof of A and deriving contradiction. But that contradiction is produced precisely by changing its mind, saying that the refutation u is mistaken because, in fact, A is the case! This time-travelling mind-changing behavior is essential to classical reasoning, and we'll see in the next lecture how this corresponds to programming with continuations.

Although it should be clear that adding either of LEM or DNE would suffice to make our logic classical, such rules violate the aesthetic principles we've adhered to thus far: both rules contain connectives, but they are neither introducing nor eliminating any single connective, and both rules contain multiple instances of a connective, suggesting a certain non-orthogonality of principles. Must our logic include negation and disjunction in order to become classical? Perhaps just negation? Or, since negation as we've seen it has been defined as implying falsehood, perhaps implication and falsehood are the important characteristics.

In fact, we can characterize what it means for a logic to be classical without appealing to any connectives at all, using purely judgmental notions. This is what we shall now endeavor to do.

4 Towards a better proof theory

Our proof theory for classical logic will be based on the idea of proof by contradiction. Proof by contradiction is closely related to double-negation elimination: If we take the rule DNE and require that its premise be proven by implication-introduction (as we know we may), we find that we could

replace it by a rule that looks like this:

$$\dfrac{\dfrac{}{\neg A\ \textit{true}}\,u \\ \vdots \\ \dfrac{\bot}{A\ \textit{true}}\,?PBC^u$$

If an assumption that $\neg A$ is true can lead to a proof of falsehood, we may conclude (classically, anyhow) that A must be true.

But we should keep in mind the design principles that keep our logics clear and easy to reason about: at most one connective should occur in a given rule, with all other notions being at the level of judgments. The proposed rule above still contains two connectives, $\neg$ and $\bot$, and does not read like an introduction or elimination rule, so we would like to replace it with a rule that appeals only to judgmental notions. We achieve this by inventing two new judgment forms, A *false* and # (contradiction):

$$\dfrac{\dfrac{}{A\ \textit{false}}\,k \\ \vdots \\ \dfrac{\#}{A\ \textit{true}}\,PBC^k$$

By convention we use letters like k and ℓ to denote hypotheses of falsity.

Now, of course, we must explain the meaning of the new judgment forms A *false* and #. We understand these judgments through certain principles analogous to the Substitution Principle we posited for hypotheses of the form A *true* before. First, we require that contradiction yield anything:

Contradiction Principle: If $\begin{matrix}\mathcal{D} \\ \#\end{matrix}$, then $\begin{matrix}\mathcal{D} \\ J\end{matrix}$.

In this principle, J stands for any judgment.

The *false* judgment is treated somewhat specially: we derive its meaning from the meaning of *true*. We take A *false* as a conclusion to be a judgment-level notational definition:

$$A\ \textit{false} := \begin{matrix}A\ \textit{true} \\ \vdots \\ \#\end{matrix}$$

To prove a proposition false, we assume it true and derive a contradiction.

We still require hypotheses of the form A *false*, since they appear in our classical rule PBC^k. To explain the meaning of hypotheses of A *false*, we derive a substitution principle from the definition above.

$$\textbf{Falsity Substitution Principle: If } \begin{array}{c} \dfrac{}{A\ \mathit{true}}\,u \\ \mathcal{D} \\ \# \end{array} \textbf{ and } \begin{array}{c} \dfrac{}{A\ \mathit{false}}\,k \\ \mathcal{E} \\ J \end{array} \textbf{ then } \begin{array}{c} \dfrac{\mathcal{D}}{A\ \mathit{false}}\,k \\ \mathcal{E} \\ J \end{array}$$

The final deduction in the above no longer depends on the hypothesis k nor on the hypothesis u.

There's an important difference between this substitution principle and the one for true hypotheses: we took that substitution principle as a given, since it arose directly from the meaning of the hypothetical judgment. Here, though, since we did not directly define A *false* as a conclusion, we are not substituting proofs that conclude A *false* for hypotheses of A *false*. Therefore, this substitution principle is one we must prove of our rules. We defer any further discussion until later when we learn rule induction.

We have one more thing to define before the system is complete, and that is how to derive contradiction! With the rules given above, there are currently no ways to do so. Contradiction means the same thing is both true and false, so we take the following rule:

$$\frac{A\ \mathit{false} \quad A\ \mathit{true}}{J}\ \mathit{contra}$$

Note that we do not suppose the rule to conclude # directly, since we wish to ensure that the Contradiction Principle above remains true. Instead, we allow the rule to conclude any judgment J, including #.

This concludes the discussion of the judgments A *false* and #. There is one remaining important technical matter to deal with, though: since we have introduced one new judgment that may appear as a conclusion, #, we must revisit any rules we previously defined that were supposed to conclude an arbitrary conclusion. In particular, the rules $\vee E^{u,v}$ and $\bot E$ must be extended to allow a conclusion of any judgment, and not merely one of the form C *true*:

$$\frac{A \vee B\ \mathit{true} \qquad \begin{array}{c} \dfrac{}{A\ \mathit{true}}\,u \\ \vdots \\ J \end{array} \qquad \begin{array}{c} \dfrac{}{B\ \mathit{true}}\,v \\ \vdots \\ J \end{array}}{J}\ \vee E^{u,v} \qquad \frac{\bot\ \mathit{true}}{J}\ \bot E$$

To show an example of these rules in action, we derive Peirce's Law, one of the characterizations of classical logic mentioned above. We show only the final deduction, but it has a flavor very similar to the derivation of $A \vee \neg A$ from DNE.

$$\dfrac{\dfrac{\dfrac{}{A\ \mathit{false}}\,k \qquad \dfrac{\dfrac{\dfrac{}{(A \supset B) \supset A\ \mathit{true}}\,u \qquad \dfrac{\dfrac{}{A\ \mathit{false}}\,k \qquad \dfrac{}{A\ \mathit{true}}\,v}{B\ \mathit{true}}\,\mathit{contra}}{A \supset B\ \mathit{true}}\,\supset I^v}{A\ \mathit{true}}\,\supset E}{\#}\,\mathit{contra}}{A\ \mathit{true}}\,PBC^k}{((A \supset B) \supset A) \supset A}\,\supset I^u$$

Note the two uses of the hypotheses k that A *false*, which is similar to the two uses of the refutation of $A \vee \neg A$ in the derivation of LEM above. Notice also the two uses of the rule contra to conclude two different judgments.

Negation revisited. Armed with our new judgmental notions, we can now give a direct definition of negation rather than one that simply defines it in terms of implication and falsehood. The new rules are as follows:

$$\dfrac{\begin{array}{c}\overline{A\ \mathit{true}}\,u \\ \vdots \\ \#\end{array}}{\neg A\ \mathit{true}}\,\neg I^u \qquad \dfrac{\neg A\ \mathit{true} \qquad \begin{array}{c}\overline{A\ \mathit{false}}\,k \\ \vdots \\ J\end{array}}{J}\,\neg E$$

The introduction rule essentially amounts to saying, "$\neg A$ *true* if A *false*," but replacing A *false* with its judgmental definition. The elimination rule lets you extract A *false* from a proof of $\neg A$ *true*, but since we only allow A *false* as a hypothesis, the rule introduces a new scope with an extra assumption rather than simply concluding A *false* directly.

This elimination rule is locally sound, as witnessed by the following local reduction, which makes use of the Falsity Substitution Principle:

$$\dfrac{\dfrac{\begin{array}{c}\overline{A\ \mathit{true}}\,u \\ \mathcal{D} \\ \#\end{array}}{\neg A\ \mathit{true}}\,\neg I^u \qquad \dfrac{\overline{A\ \mathit{false}}\,k \qquad \mathcal{E}}{J}}{J}\,\neg E^k \quad \Longrightarrow_R \quad \dfrac{\mathcal{D}}{\dfrac{A\ \mathit{false}}{J}\,k \quad \mathcal{E}}$$

It is also locally complete; we can expand an arbitrary deduction of $\neg A\ true$ in two ways:

$$\begin{array}{c} \mathcal{D} \\ \neg A\ true \end{array} \Longrightarrow_E \dfrac{\begin{array}{c}\mathcal{D}\\ \neg A\ true\end{array} \qquad \dfrac{\dfrac{\dfrac{}{A\ false}\,k \quad \dfrac{}{A\ true}\,u}{\#}\,contra}{\neg A\ true}\,\neg I^u}{\neg A\ true}\,\neg E^k$$

$$\begin{array}{c} \mathcal{D} \\ \neg A\ true \end{array} \Longrightarrow_E \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \neg A\ true\end{array} \qquad \dfrac{\dfrac{\dfrac{}{A\ false}\,k \quad \dfrac{}{A\ true}\,u}{\#}\,contra}{\#}\,\neg E^k}{\#}\,\neg I^u}{\neg A\ true}$$

Note how the elim rule can be used to conclude either # or $\neg A\ true$.

To illustrate the use of negation, we give one final example deduction, a proof of the inverse contrapositive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B \supset \neg A\ true}\,u \qquad \dfrac{\dfrac{\dfrac{}{B\ false}\,k \quad \dfrac{}{B\ true}\,w}{\#}\,contra}{\neg B\ true}\,\neg I^w}{\neg A\ true}\,\supset E \qquad \dfrac{\dfrac{}{A\ false}\,\ell \quad \dfrac{}{A\ true}\,v}{\#}\,contra}{\#}\,\neg E^\ell}{\dfrac{\dfrac{\#}{B\ true}\,PBC^k}{A \supset B\ true}\,\supset I^v}{(\neg B \supset \neg A) \supset A \supset B\ true}\,\supset I^u$$

The rules for classical logic are summarized in Figure 1. It is worth noting that the only rule which makes this logic actually classical is the rule of proof-by-contradiction, PBC^k. All of the other rules, including the new, direct explanation of negation, are perfectly valid from an intuitionistic perspective: intuitionists still deal with falsehood and contradiction, just in a more controlled fashion.

Judgmental Definitions

$$A \textit{ false} := \begin{array}{c} A \textit{ true} \\ \vdots \\ \# \end{array} \qquad \frac{A \textit{ false} \quad A \textit{ true}}{J}\ \textit{contra}$$

Classical Rule

$$\dfrac{\dfrac{}{A \textit{ false}}\ k \\ \vdots \\ \dfrac{\#}{A \textit{ true}}\ PBC^k}$$

Rules for Negation

$$\dfrac{\dfrac{}{A \textit{ true}}\ u \\ \vdots \\ \dfrac{\#}{\neg A \textit{ true}}\ \neg I^u} \qquad \dfrac{\neg A \textit{ true} \quad \begin{array}{c} \dfrac{}{A \textit{ false}}\ k \\ \vdots \\ J \end{array}}{J}\ \neg E$$

Figure 1: Rules for classical natural deduction

Lecture Notes on Classical Computation

15-317: Constructive Logic
Ronald Garcia

Lecture 8
September 17, 2009

1 Introduction

In the last lecture, we talked about how to alter our system of logic to support classical reasoning. To do so, we introduced two new judgments: # signifying contradiction, and $A\ \mathit{false}$. In this lecture, we explore a computational interpretation of this new system.

2 Proof Terms

We begin by associating proof terms with each of our new rules. The rule of contradiction follows:

$$\frac{k : A\ \mathit{false} \quad M : A\ \mathit{true}}{\mathbf{throw}^J\ k\ M : J}\ \mathit{contra}$$

Since contradiction can produce any judgment whatsoever, we annotate the proof term constructor as $\mathbf{throw}^J$ so that it precisely captures the proof. We will omit this annotation when it's clear from the context.

The proof term assignment for proof by contradiction is as follows:

$$\dfrac{\dfrac{}{k : A\ \mathit{false}}\ k \\ \vdots \\ E : \#}{\mathcal{C}k.E : A\ \mathit{true}}\ PBC^k$$

As with proof terms for implication introduction, we annotate the proof term variable as $\mathcal{C}k{:}A.E$ to disambiguate the proof.

We will study these two rules and their associated proof terms to see how classical logic corresponds to computation.

As you know from last time, $A \vee \neg A$, often called the law of the excluded middle (LEM), is true in classical logic but not in intuitionistic logic. Here is a proof of LEM annotated with terms:

$$\dfrac{\dfrac{}{k : A \vee \neg A \textit{ false}}\,k \quad \dfrac{\dfrac{}{v : A \textit{ true}}\,v}{\textbf{inl}\ v : A \vee \neg A \textit{ true}}\,\vee I}{\textbf{throw}\ k\ (\textbf{inl}\ v) : \bot \textit{ true}}\,\textit{contra}$$

$$\dfrac{\dfrac{}{k : A \vee \neg A \textit{ false}}\,k \quad \dfrac{\dfrac{\textbf{throw}\ k\ (\textbf{inl}\ v) : \bot \textit{ true}}{\lambda v.\textbf{throw}\ k\ (\textbf{inl}\ v) : \neg A \textit{ true}}\,\supset I^{v}}{\textbf{inr}\ (\lambda v.\textbf{throw}\ k\ (\textbf{inl}\ v)) : A \vee \neg A \textit{ true}}\,\vee I}{\textbf{throw}\ k\ (\textbf{inr}\ (\lambda v.\textbf{throw}\ k\ (\textbf{inl}\ v))) : \#}\,\textit{contra}$$

$$\dfrac{\textbf{throw}\ k\ (\textbf{inr}\ (\lambda v.\textbf{throw}\ k\ (\textbf{inl}\ v))) : \#}{\mathcal{C}k.\textbf{throw}\ k\ (\textbf{inr}\ (\lambda v.\textbf{throw}\ k\ (\textbf{inl}\ v))) : A \vee \neg A \textit{ true}}\,PBC^{k}$$

In this lecture $\neg A$ is once again notation for $A \supset \bot$. Observe that v is a proof of A *true*, while $\lambda v.\textbf{throw}\ k\ (\textbf{inl}\ v)$ is a proof of $\neg A$ *true*. Notice as well that **throw** is used to produce two separate judgments. Once it yields $\bot$ *true*, which we need to produce a proof of $\neg A$ *true*, and once it yields a contradiction #, which is used in a proof by contradiction. Finally, notice that here we're using *judgments as types* not just propositions as types. Since our proof terms represent more judgments than A *true*, it's not sufficient to simply give A as the type: we must capture whether A is true or false, or if we have produced a contradiction.

Figure 1 proves that $(A \supset B \vee C) \supset (A \supset B) \vee (A \supset C)$ using excluded middle as a lemma:

$$\mathit{LEM} = \mathcal{C}k{:}A \vee \neg A.\textbf{throw}^{\#}\ k\ (\textbf{inr}^{A}\ (\lambda v{:}A.\textbf{throw}^{\bot}\ k\ (\textbf{inl}^{\neg A}\ v)))$$

The proof term that corresponds to this proof is as follows:

$$\begin{array}{l}
\lambda f.\textbf{case}\ \mathit{LEM}\ \textbf{of} \\
\quad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda_.u) \\
\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda_.w) \\
\quad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))
\end{array}$$

For a final example, consider Peirce's Law, $((A \supset B) \supset A) \supset A$.
In a previous class, we tried to write a function with this type, but found

$$\dfrac{\dfrac{}{LEM : A \vee \neg A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \supset B \vee C}\, f \quad \dfrac{}{A}\, x}{B \vee C} \supset E}{(A \supset B) \vee (A \supset C)} \vee I \quad \dfrac{\dfrac{\dfrac{}{B}\, u}{A \supset B} \supset I\text{-}}{(A \supset B) \vee (A \supset C)} \vee I \quad \dfrac{\dfrac{\dfrac{}{C}\, w}{A \supset C} \supset I\text{-}}{(A \supset B) \vee (A \supset C)} \vee I}{(A \supset B) \vee (A \supset C)} \vee E^{u,w} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\neg A}\, y \quad \dfrac{}{A}\, z}{\bot} \supset E}{B} \bot E}{A \supset B} \supset I^{z}}{(A \supset B) \vee (A \supset C)} \vee I}{(A \supset B) \vee (A \supset C)} \vee E^{x,y}$$

$$\dfrac{(A \supset B) \vee (A \supset C)}{(A \supset B \vee C) \supset (A \supset B) \vee (A \supset C)} \supset I^{f}$$

Figure 1: Classical proof using the law of the excluded middle.

that it was not possible because it's not intuitionistically true. Now using classical logic, we can both prove it and program it!

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A\ \mathit{false}}\, k \quad \dfrac{}{A\ \mathit{true}}\, u}{\boxed{\mathrm{B}}}\ \mathit{contra}}{A\ \mathit{true}} \supset E}{\dfrac{\dfrac{}{A\ \mathit{false}}\, k \quad \dfrac{}{(A \supset B) \supset A\ \mathit{true}}\, f}{A\ \mathit{true}}\ \mathit{contra}}{A \supset B\ \mathit{true}} \supset I^{u}$$

$$\dfrac{\dfrac{\#}{A\ \mathit{true}}\ PBC^{k}}{((A \supset B) \supset A) \supset A\ \mathit{true}} \supset I^{f}$$

Observe the boxed B. The contradiction rule concludes B *true* out of thin air and uses it to conclude that $A \supset B$ *true* thereby discharging our assumption of A *true*. We can then use this $A \supset B$ *true* to eliminate $((A \supset B) \supset A) \supset A$ *true* and conclude A without ever having a real proof of A! Then we use proof by contradiction to discharge our assumption that A *false*. The proof term for Pierce's law is:

$$\lambda f.\mathcal{C}k.\mathbf{throw}\ k\ (f\ (\lambda u.\mathbf{throw}\ k\ u))$$

Notice that it **throw**s k twice: once to produce a type that we need out of thin air, (via contradiction) and once to judge # so that the variable k can be bound using $\mathcal{C}$. The proof of LEM does the same thing. This is a common pattern for proofs (and programs) that depend on classical logic.

3 Reduction

Adding proof by contradiction to our logic changes the meaning of all of our connectives because there are now new ways to introduce each of them: the introduction rules are no longer the sole means.

Having changed our system, it's now necessary to check that local soundness and local completeness still hold for each connective. Local completeness still holds exactly the same way: given a proof of say $A \wedge B$, one can still use the original elimination and introduction rules to perform a local expansion.

On the other hand, local soundness must be shown to apply for every combination of introduction and elimination rules. In our new system, proof by contradiction can be used to introduce every propositional connective, so we must show that proof-by-contradiction followed by an elimination can be reduced.

Looking at the rules of the system, it's not necessarily obvious how one could reduce a proposition introduced by contradiction. Let's demonstrate how using implication as an example:

$$\dfrac{\dfrac{\dfrac{}{A \supset B\ \mathit{false}}\ k \quad \overset{\mathcal{D}}{A \supset B\ \mathit{true}}}{\#}\ \mathit{contra}}{\dfrac{\dfrac{\#}{A \supset B\ \mathit{true}}\ PBC^k \qquad \overset{\mathcal{E}}{A\ \mathit{true}}}{B\ \mathit{true}}\ \supset E} \quad \Longrightarrow_R$$

$$\dfrac{\dfrac{}{B\ \mathit{false}}\ k' \quad \dfrac{\overset{\mathcal{D}}{A \supset B\ \mathit{true}} \quad \overset{\mathcal{E}}{A\ \mathit{true}}}{B\ \mathit{true}}\ \supset E}{\#}\ \mathit{contra}}{B\ \mathit{true}}\ PBC^{k'}$$

Observe how the local reduction behaves. The PBC rule introduced an implication $A \supset B$ *true*, which was immediately eliminated. To reduce this rule, the elimination rule is pushed up to the point in the proof where the contra rule is applied to the assumption $A \supset B$ *false* which was labeled k. Furthermore, the contra rule is now applied to an assumption B *false* which is now labeled with k', and the PBC rule now discharges this new assumption (the variable name doesn't have to change, but it can; from here on we often keep it the same).

Referring to the corresponding proof terms, the reduction looks like the following:

$$(\mathcal{C}k.\textbf{throw}\ k\ M)\ N \Longrightarrow_R \mathcal{C}k.\textbf{throw}\ k\ (M\ N)$$

Here, M is the proof term for $\mathcal{D}$ and N is the proof term for $\mathcal{E}$.

Local reduction operates similarly in the case of conjunction:

$$\textbf{snd}\ (\mathcal{C}k.\textbf{throw}\ k\ M) \Longrightarrow_R \mathcal{C}k.\textbf{throw}\ k\ (\textbf{snd}\ M)$$

In this case, M is a proof term for $A \wedge B$ *true*, and the entire term is a proof of B *true*.

The two examples above can be generalized to describe how local reductions involving $\mathcal{C}k.E$ behave. Each elimination rule is associated with some operation: **fst** and **snd** for conjunction, **case** for disjunction, and **abort** for false. Whenever one of these operations is applied to a proof by contradiction, the reduction rule "steals" the operation and copies it to every location where the abstracted variable k is thrown in its body. For example:

$$\textbf{snd}\ (\mathcal{C}k.E) \Longrightarrow_R \mathcal{C}k'.[k'\ (\textbf{snd}\ \Box)/k]E$$

The syntax $[k'\ (\textbf{snd}\ \Box)/k]E$ stands for a new kind of substitution, *structural substitution* into the body of E. Roughly speaking, in the body of E, every instance of **throw** $k\ M$ is replaced with **throw** $k'\ (\textbf{snd}\ M)$. The box $\Box$ stands for the "hole" where the old argument to throw, M, gets placed. The label k' corresponds to the proposition proved by the elimination rule.

The full set of rules are as follows:

$$\begin{array}{lll}
(\mathcal{C}k.\textbf{throw}\ k\ M)\ N & \Longrightarrow_R & \mathcal{C}k.\textbf{throw}\ k\ (M\ N) \\
\textbf{fst}\ (\mathcal{C}k.E) & \Longrightarrow_R & \mathcal{C}k'.[k'\ (\textbf{fst}\ \Box)/k]E \\
\textbf{snd}\ (\mathcal{C}k.E) & \Longrightarrow_R & \mathcal{C}k'.[k'\ (\textbf{snd}\ \Box)/k]E \\
\textbf{case}\ (\mathcal{C}k.E)\ \textbf{of} & & \\
\quad \textbf{inl}\ u => M & \Longrightarrow_R & \mathcal{C}k'.\left[\begin{array}{l}\textbf{case}\ \Box\ \textbf{of} \\ \quad \textbf{inl}\ u => M \\ \quad \textbf{inr}\ v => N\end{array}\Big/\ k\right] E \\
\quad \textbf{inr}\ v => N & & \\
\textbf{abort}\ (\mathcal{C}k.E) & \Longrightarrow_R & \mathcal{C}k'.[k'\ (\textbf{abort}\ \Box)/k]E
\end{array}$$

Not only can proof-by-contradiction introduce any logical connective, but plain ole' contradiction, which we associate with the **throw** operator, can as well! As we saw earlier, this was critical in proving Peirce's law and the law of the excluded middle. It turns out that a contradiction followed by an elimination can also be reduced. Here is an example using implica-

tion:

$$\dfrac{\dfrac{\dfrac{}{C\ \mathit{false}}\ k \quad \dfrac{\mathcal{D}}{C\ \mathit{true}}}{A \supset B\ \mathit{true}}\ \mathit{contra} \quad \dfrac{\mathcal{E}}{A\ \mathit{true}}}{B\ \mathit{true}}\ \supset E \quad \Longrightarrow_R$$

$$\dfrac{\dfrac{}{C\ \mathit{false}}\ k \quad \dfrac{\mathcal{D}}{C\ \mathit{true}}}{B\ \mathit{true}}\ \mathit{contra}$$

In this case, the local reduction *annihilates* the elimination rule, and now the *contra* rule simply concludes what the elimination rule used to. Here is the reduction written using proof terms:

$$(\mathbf{throw}^{A \supset B}\ k\ M)\ N \Longrightarrow_R (\mathbf{throw}^{B}\ k\ M)$$

Just like for $\mathcal{C}$, this reduction generalizes to the other elimination rules:

$$\begin{array}{lcl}
\mathbf{fst}\ (\mathbf{throw}^{A \wedge B}\ k\ M) & \Longrightarrow_R & \mathbf{throw}^{A}\ k\ M \\
\mathbf{snd}\ (\mathbf{throw}^{A \wedge B}\ k\ M) & \Longrightarrow_R & \mathbf{throw}^{B}\ k\ M \\
\mathbf{case}\ (\mathbf{throw}^{A \vee B}\ k\ M)\ \mathbf{of} & & \\
\quad \mathbf{inl}\ u => M & \Longrightarrow_R & \mathbf{throw}^{C}\ k\ M \\
\quad \mathbf{inr}\ v => N & & \\
\mathbf{abort}^{C}\ (\mathbf{throw}^{\bot}\ k\ M) & \Longrightarrow_R & \mathbf{throw}^{C}\ k\ M
\end{array}$$

In each case, **throw** eats any attempt to eliminate it, and like a chameleon, dresses itself up to look like what it should have been eliminated into! In the case of disjunction elimination, C is the proposition that is proved by both M and N.

To see these reductions in action, let's revisit our proof that $(A \supset B \vee C) \supset (A \supset B) \vee (A \supset C)$. To keep things somewhat manageable, we will only consider reductions on its proof term which is:

$$\begin{array}{l}
\lambda f.\mathbf{case}\ \mathit{LEM}\ \mathbf{of} \\
\quad \mathbf{inl}\ x \Rightarrow \mathbf{case}\ f\ x\ \mathbf{of} \\
\qquad\qquad \mathbf{inl}\ u \Rightarrow \mathbf{inl}\ (\lambda_.u) \\
\qquad\qquad \mathbf{inr}\ w \Rightarrow \mathbf{inr}\ (\lambda_.w) \\
\quad \mathbf{inr}\ y \Rightarrow \mathbf{inl}\ (\lambda z.\mathbf{abort}(y\ z))
\end{array}$$

where

$$\mathit{LEM} = (\mathcal{C}k.\mathbf{throw}\ k\ (\mathbf{inr}\ (\lambda v.\mathbf{throw}\ k\ (\mathbf{inl}\ v))))$$

We can perform local reductions wherever a rule applies. We'll try to be systematic, working our way from the outside in. First, the **case** operator is eliminating the $\mathcal{C}$ inside of LEM, which introduces $A \vee \neg A$, so we can use our "steal and copy" reduction rule:

$$\begin{array}{l}
\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inr}\ (\lambda v.\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inl}\ v)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda_.u) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda_.w) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda_.u) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda_.w) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))
\end{array}$$

The **case** operation has been absorbed into the $\mathcal{C}$ operator, and wrapped around the second argument to both throw expressions.

Now, the second argument to the first **throw** is a **case** applied to an **inr**. Recall that before this **case** was "stolen", it was eliminating the law of the excluded middle, $A \vee \neg A$, and here it finds out that our evidence is *really* (wink wink) a proof of $\neg A$, since it's wrapped in an **inr**. We can reduce this disjunction introduction (**inr**) and elimination (**case**) by substituting the content of the **inr** into the corresponding branch of the **case** expression:

$$\begin{array}{l}
\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k \\
\quad \textbf{inl}\ (\lambda z.\textbf{abort}((\lambda v.\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inl}\ v)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda_.u) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda_.w) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))))\ z))
\end{array}$$

Next, in the argument to **abort**, $\lambda v...$ is applied to z so we can substitute z for v:

$$\begin{array}{l}
\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k \\
\quad \textbf{inl}\ (\lambda z.\textbf{abort}(\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inl}\ z)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda_.u) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda_.w) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z)))))
\end{array}$$

Now there is an **abort** applied to a **throw**. According to our new local reductions, **throw** can eat **abort** (how tragic!):

$$\begin{array}{l}\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k \\ \quad \textbf{inl}\ (\lambda z.\textbf{throw}\ k\ (\textbf{case}\ (\textbf{inl}\ z)\ \textbf{of} \\ \qquad\qquad\qquad \textbf{inl}\ x \Rightarrow \textbf{case}\ f\ x\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\ \qquad\qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w) \\ \qquad\qquad\qquad \textbf{inr}\ y \Rightarrow \textbf{inl}\ (\lambda z.\textbf{abort}(y\ z))))\end{array}$$

Now we're back at the *same* **case** statement that we started with! It seems like we've gone back in time and have to resolve this case all over again, but this time the **case** expression is eliminating an **inl**, i.e. a proof of A! We can reduce it using the same strategy, but substituting into the other branch of the case statement:

$$\begin{array}{l}\Longrightarrow_R \lambda f.\mathcal{C}k.\textbf{throw}\ k \\ \quad \textbf{inl}\ (\lambda z.\textbf{throw}\ k\ \textbf{case}\ f\ z\ \textbf{of} \\ \qquad\qquad\qquad\qquad \textbf{inl}\ u \Rightarrow \textbf{inl}\ (\lambda\_.u) \\ \qquad\qquad\qquad\qquad \textbf{inr}\ w \Rightarrow \textbf{inr}\ (\lambda\_.w)\end{array}$$

At this point, we've performed all of the local reductions that we can. What are we left with? Figure 2 is the proof tree corresponding to this term. It's a proof of the same theorem, $(A \supset B \vee C) \supset (A \supset B) \vee (A \supset C)$ *true*, but this one uses proof by contradiction directly instead of assuming the law of the excluded middle.

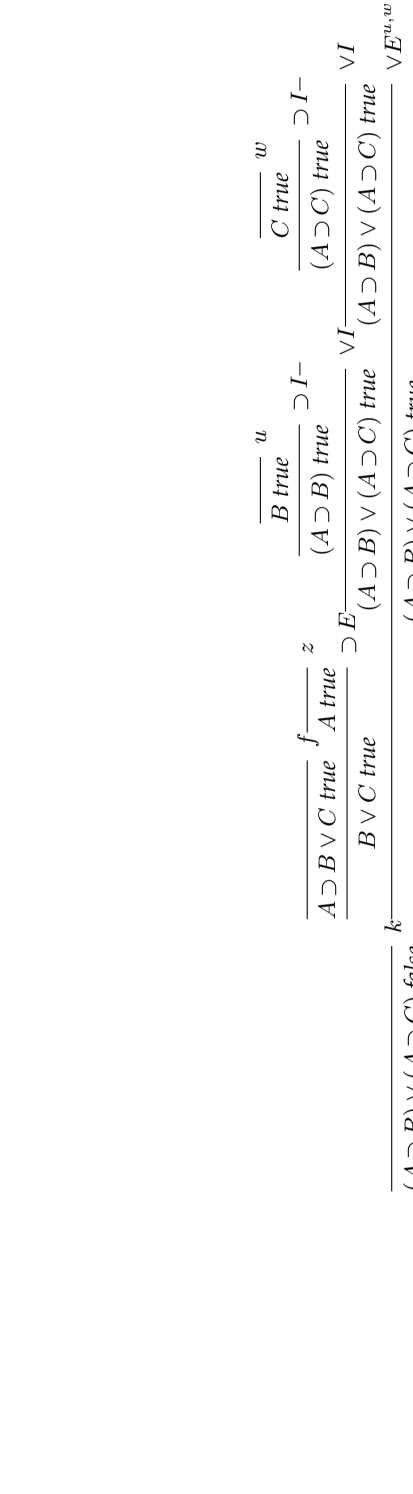


Figure 2: Direct proof of $(A \supset B \vee C) \supset (A \supset B) \vee (A \supset C)$ *true*

Looking at how we arrived at this proof, you can see how the classical operators:

1. lie about what they are proofs of;
2. steal any attempt to apply elimination rules to them; and
3. send you through time warps so you can repeat the same work but make different choices each time.

4 Relating Classical Logic to Intuitionistic Logic

Let's write $\Gamma \vdash_c A$ *true* for classical truth and $\Gamma \vdash_i A$ *true* for intuitionistic truth, where we put a context Γ of hypotheses in the judgment form rather than using the two-dimensional notation for hypotheses.

It's easy to prove that:

If $\Gamma \vdash_i A$ *true* then $\Gamma \vdash_c A$ *true*.

This says that if an intuitionist asserts A, a classicist will believe him, interpreting A classically. Informally, the move from intuitionistic to classical logic consisted of adding new inference rules, so whatever is true intuitionistically must be true classically. This can be formalized as a proof by rule induction on $\Gamma \vdash_i A$ *true*.

Of course, the opposite entailment does not hold (take A to be the law of the excluded middle, or double-negation elimination). However, it is possible to *translate* propositions in such a way that, if a proposition is classically true, then its translation is intuitionistically true. That is, the intuitionist does not believe what the classicist says at face value, but he can figure out what the classicist really meant to say, by means of a *double-negation translation*. The translation inserts enough double-negations into the proposition A that the classical uses of the DNE rule are intuitionistically permissible.

We will use the "Gödel-Gentzen negative translation", which is defined by a function $A^* = A'$ from classical propositions to intuitionistic propositions. On the intuitionistic side, we use the usual notational definition of $\neg A = (A \supset \bot)$.

$$\begin{aligned}
(\top)^* &= \top \\
(\bot)^* &= \bot \\
(A \wedge B)^* &= A^* \wedge B^* \\
(A \vee B)^* &= \neg\neg(A^* \vee B^*)
\end{aligned}$$

$$\begin{aligned}(A \supset B)^* &= (A^* \supset B^*)\\ (\neg A)^* &= \neg A^*\\ (P)^* &= \neg\neg P\end{aligned}$$

That is, the classicist and the intuitionistic agree about the meaning of all of the connectives except $\vee$ and atomic propositions P. From an intuitionistic point of view, when a classicist says $A \vee B$, he *really* means $\neg\neg(A^* \vee B^*)$, an intuitionistically weaker statement. Thus, **intuitionistic logic is more precise**, because you can say $A \vee B$, if that's the case, or $\neg\neg(A \vee B)$ if you need classical reasoning to do the proof. There is no way to express intuitionistic disjunction in classical logic. If an intuitionist says A to a classicist, and then the classicist repeats it back to him, it will come back as a weaker statement A^*.

On the other hand, the translation has the property that A and A^* are classically equivalent. If a classicist says something to an intuitionist, and then the intuitionist repeats it back to him, the classicist won't know the difference: intuitionistic logic makes finer distinctions.

As an aside, there are several other ways of translating classical logic into intuitionistic logic, which make different choices about where to insert double-negations. Different translations do different things to proofs, which turns out to have interesting consequences for programming.

The following statement captures what we mean when we say that this translation "does the right thing".

$\Gamma \vdash_c J$ iff $\Gamma^* \vdash_i J^*$.

5 Programming with Classical Logic: Continuations

Earlier we showed the proof term for Peirce's Law:

$$\lambda f.\mathcal{C}k.\textbf{throw}\ k\ (f\ (\lambda u.\textbf{throw}\ k\ u))$$

This proof of $((A \supset B) \supset A) \supset A$ corresponds to a powerful programming language construct. Peirce's law is the type of "call with current continuation" or `callcc`, a powerful operator that appears in the Scheme programming language and in Standard ML of New Jersey. Thinking operationally, `callcc` gives you a copy of the current call stack of a program that you can hold on to. Then later, after the program has done some more work

and the stack has changed, you can replace the current stack with your old stack, in a sense turning back the sands of time to an old program state.

The `callcc` operator has an immediately-binding variant called **letcc**:

$$\textbf{letcc}(k.M) \equiv \mathcal{C}k.\textbf{throw}\ k\ M$$

With `callcc` in Scheme, the **throw** operation is wrapped up inside of a function $(\lambda u.\textbf{throw}\ k\ u)$, so you can just call the continuation like any other function. The **letcc** operator requires you to explicitly **throw** the continuation k, just like with the $\mathcal{C}$ operator. Notice that both `callcc` and **letcc** immediately throw the continuation that they capture.

As an example of programming with continuations, consider a function that multiples all the integers in a list. In writing this code, we'll assume that `intlist` and `int` are *propositions*, like they would be in ML, and that we can write pattern-matching functions over them. Here's a first version:

```
mult' : intlist => int
mult' [] = 1
mult' (x :: xs) = x * mult' xs
```

The multiplication of the empty list is `1`, and the multiplication of `x :: xs` is the head times the multiplication of the tail.

What happens when we call `mult' [1,2,3,0,4,5,....]` where the `...` is 700 billion[1] more numbers? It does a lot more work than necessary to figure out that the answer is 0. Here's a better version:

```
mult' : intlist => int
mult' [] = 1
mult' (0 :: xs) = 0
mult' (x :: xs) = x * mult' xs
```

This version checks for 0, and returns 0 immediately, and therefore does better on the list `[1,2,3,0,4,5,....]`.

But what about the reverse list `[...,5,4,0,1,2,3]`? This version still does all 700 billion multiplications on the way up the call chain, which could also be skipped.

We can do this using continuations:

```
mult xs = letcc k in
  let
```

[1] this week's trendy really-large number to pull out of thin air

```
  mult' : intlist => int
  mult' [] = 1
  mult' (0 :: xs) = throw k 0
  mult' (x :: xs) = x * (mult' xs)
in throw k (mult' xs)
```

The idea is that we grab a continuation `k` standing for the evaluation context in which `mult` is called. Whenever we find a 0, we **immediately** jump back to this context, with no further multiplications. If we make it through the list without finding a zero, we throw the result of `mult'` to this continuation, returning it from `mult`. Note that we could have just returned this value since `letcc k` has a `throw k` built-in.

In this program, continuations have been used simply to jump to the top of a computation. Other more interesting programs use continuations to jump out and then back in to a computation, resuming where you left off.

6 Continuation-Passing Style: double-negation translation for Programs

Continuations are a powerful programming language feature for classical functional programs, but what if you only have an intuitionistic functional programming language? Well, recall that there are several kinds of double-negation transformations which can embed classical logic propositions into intuitionistic logic. It turns out that this same strategy can be applied to programs: we can translate classical programs into intuitionistic programs. The process is called a *continuation-passing style* transformation, or CPS for short, and just as there are several different double-negation translations, there are several different CPS's.

For the moment, let's limit ourselves to a logic with only implication $\supset$ and falsehood $\bot$. Another translation $A^\star$ is defined as follows:

$$\begin{array}{lcl} P^\star & = & P \\ (A \supset B)^\star & = & (A^\star \supset \neg\neg B^\star) \end{array}$$

and has the property that A holds classically iff $\neg\neg A^\star$ intuitionistically.

A corresponding program translation $\overline{M}$ takes a program of type A to a

program of type $(A^\star \supset \bot) \supset \bot$:

$$\begin{array}{lcl}
\overline{x} & = & \lambda k.k\ x \\
\overline{\lambda x.M} & = & \lambda k.k\ (\lambda x.\overline{M}) \\
\overline{M\ N} & = & \lambda k.\overline{M}(\lambda f.\overline{N}(\lambda x.f\ x\ k) \\
\overline{\textbf{letcc}\ k_0.M} & = & (\lambda k.\overline{M}\ k)[(\lambda a.\lambda k.k\ a)/k_0] \\
\overline{\textbf{throw}\ k_0\ M} & = & \overline{k_0\ M}
\end{array}$$

So CPS will turn a program of type `intlist -> int` into a program of type `(intlist -> (int -> 'o) -> 'o) -> 'o -> 'o` where `'o` is some type that plays the role of $\bot$. As it turns out, `'o` ends up being the type of the whole program.

To give some flavor for this translation, let's look at our mult function after CPS:

```
mult-k : intlist -> (int -> 'o) -> 'o
mult-k xs k0 =
  let k = k0 in
  let
   mult-k' : intlist -> (int -> int) -> int
   mult-k' [] k1 = k1 1
   mult-k' (0 :: xs) k1 = k 0
   mult-k' (x :: xs) k1 = (mult-k' xs (fn v => k1 (x * v)))
  in (mult-k' xs k)

mult-cps : (intlist -> (int -> 'o) -> 'o) -> 'o -> 'o
mult-cps = fn k => k mult
```

The functions above are a simplified version of the output of CPS, so there are not as many `fn k =>...` functions as would come out of the literal translation. The function `mult-cps` is the CPS counterpart of the original `mult` function.

To simplify matters, we'll work directly with `mult-k`. In contrast to the original `mult` function, this one takes an extra argument, `k0` of type `int -> 'o`. This is a function-based representation of the current continuation, all of the work that's left to be done. The first line in the function stores a copy of this continuation in a variable `k`. This is the counterpart to `letcc` from the classical program.

Now instead of throwing k, we can simply call the function `k` and pass it a value. Notice how everywhere in `mult-k'` that used to simply return a value, it now passes that value to the current continuation. Also, where it

used to say `x * (mult' xs)` now calls the function `mult-k'` and passes it a *bigger* continuation that accepts a value `v`, the result of `mult-k' xs`, multiplies that value by `x`, then calls the continuation `k1`. This is how the stack grows. Any time there used to be more work to do after a function call returns, it has now been rolled into the continuation that is passed to that function.

Finally notice that the `0` case of `mult-k'` ignores its current continuation `k1` and instead calls the continuation that was captured at the top of `mult-k`. So how do we get a value back from this program? Well, we can pass it a continuation that simply returns whatever it gets:

```
id : int -> int
id x = x

mult-k big-list id
```

Now we've substituted the type `int` for the indeterminate type `'o`, and when the computation is completed, it will call this initial continuation `id` which returns the answer.

The CPS transformation is not simply an academic exercise. Some compilers use CPS internally while translating programs into executables. This makes it really easy to provide support for language features like **letcc** and similar operators.

References

[AHS07] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A proof-theoretic foundation of abortive continuations. *Higher-Order and Symbolic Computation*, 20(4):403–429, December 2007.

[Dyb09] R. Kent Dybvig. *The Scheme Programming Language*, chapter 3.3-3.4. Prentice-Hall, Englewood Cliffs, New Jersey, fourth edition, 2009. Available electronically at `http://www.scheme.com/tspl4`.

[FHK84] D. P. Friedman, C. T. Haynes, and E. Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformation and Programming Environments*, NATO ASI Series F V8, pages 263–274. Springer Verlag, 1984.

[FW08] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*, chapter 6: Continuation-Passing Style. MIT Press, Cambridge, MA, third edition, 2008.

[Wan99] Mitchell Wand. Continuation-based multiprocessing. *Higher-Order and Symbolic Computation*, 12(3):285–299, October 1999.

Lecture Notes on Sequent Calculus

15-317: Constructive Logic
Frank Pfenning

Lecture 9
September 24, 2009

1 Introduction

In this chapter we develop the sequent calculus as a formal system for proof search in natural deduction. The 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 3. 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 an assumption may be used as a verification for an atomic formula. In the sequent calculus, both steps work bottom-up, which will allows us to prove global versions of the local soundness and completeness properties foreshadowed in Lecture 3.

2 Sequents

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

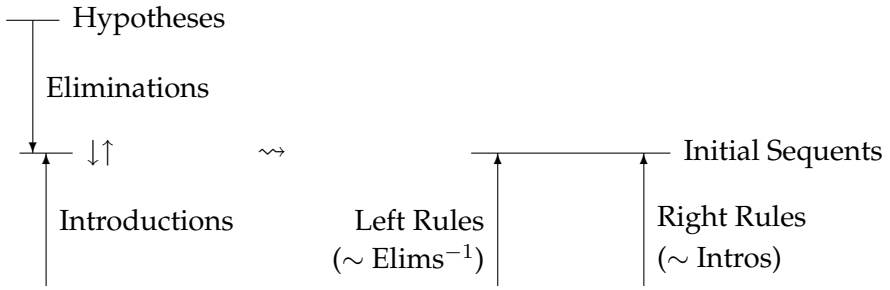
$$\begin{array}{c} A_1{\downarrow} \quad \cdots \quad A_n{\downarrow} \\ \vdots \\ C{\uparrow} \end{array}$$

where $A_1, \ldots, A_n$ are all assumptions 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

$$A_1 \text{ left}, \ldots, A_n \text{ left} \Longrightarrow C \text{ right}$$

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 called the *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 left, $\ldots, A_n$ left by Γ. 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.

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

The only difference is that the antecedents Γ 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 two corresponding left rules. Since the letters L and R are used to denote the type of rule in the sequent calculus, we index the rules as first and second conjunction left rule.

$$\frac{A \wedge B\downarrow}{A\downarrow}\ \wedge E_L \qquad \frac{\Gamma, A \wedge B, A \Longrightarrow C}{\Gamma, A \wedge B \Longrightarrow C}\ \wedge L_1$$

$$\frac{A \wedge B\downarrow}{B\downarrow}\ \wedge E_R \qquad \frac{\Gamma, A \wedge B, B \Longrightarrow C}{\Gamma, A \wedge B \Longrightarrow C}\ \wedge L_2$$

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.

$$\frac{}{\top\uparrow}\ \top I \qquad \frac{}{\Gamma \Longrightarrow \top}\ \top R$$

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

$$\frac{\begin{array}{c}\overline{A\downarrow}\ u \\ \vdots \\ B\uparrow\end{array}}{A \supset B\uparrow}\ \supset^u \qquad \frac{\Gamma, A \Longrightarrow B}{\Gamma \Longrightarrow A \supset B}\ \supset R$$

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 rule 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.

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

In words: in order to use $A \supset B$ we have to produce a verification of A, in which case we can use B. 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.

$$\frac{A\uparrow}{A \vee B\uparrow}\ \vee I_L \qquad \frac{\Gamma \Longrightarrow A}{\Gamma \Longrightarrow A \vee B}\ \vee R_1$$

$$\frac{B\uparrow}{A \vee B\uparrow}\ \vee I_R \qquad \frac{\Gamma \Longrightarrow B}{\Gamma \Longrightarrow A \vee B}\ \vee R_2$$

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.

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

As for implication, scoping issues are simplified because the new assumptions A and B in the first and second premise, respectively, are available anywhere in the deduction above.

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.

$$\frac{\bot\downarrow}{C\uparrow}\ \bot E \qquad \frac{}{\Gamma, \bot \Longrightarrow C}\ \bot L$$

Atomic propositions. 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. This turns into a so-called *initial sequent*.

$$\frac{P\downarrow}{P\uparrow}\ \downarrow\uparrow \qquad \frac{}{\Gamma, P \Longrightarrow P}\ \mathsf{init}$$

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 $(A \vee B) \supset (B \vee A)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vee B, A \Longrightarrow A}\ \mathsf{init}}{A \vee B, A \Longrightarrow B \vee A}\ \vee R_2 \quad \dfrac{\dfrac{}{A \vee B, B \Longrightarrow B}\ \mathsf{init}}{A \vee B, B \Longrightarrow B \vee A}\ \vee R_1}{A \vee B \Longrightarrow B \vee A}\ \vee L}{\Longrightarrow (A \vee B) \supset (B \vee A)}\ \supset R$$

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 $\Longrightarrow A$), 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* $\Longrightarrow \bot$.

Proof: No left rule is applicable, since there is no antecedent. No right rule is applicable, because there is no right rule for falsehood. Therefore, there cannot be a proof of $\Longrightarrow \bot$. □

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

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 $\Longrightarrow A$, in the second $\Longrightarrow B$. □

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

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

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* $\Gamma \Longrightarrow C$ *then* $\Gamma, A \Longrightarrow C$ *with a structurally identical deduction.*

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

Theorem 5 (Contraction) *If* $\Gamma, A, A \Longrightarrow C$ *then* $\Gamma, A \Longrightarrow C$ *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. □

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

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

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

$$\frac{\neg\neg A \Longrightarrow \neg A \quad \neg\neg A, \bot \Longrightarrow A}{\neg\neg A \Longrightarrow A}\ \supset L$$

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.

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

$$\frac{\neg\neg A, A \Longrightarrow \bot}{\neg\neg A \Longrightarrow \neg A}\ \supset R$$

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

$$\frac{\neg\neg A, A \Longrightarrow \neg A \quad \neg\neg A, A, \bot \Longrightarrow \bot}{\neg\neg A, A \Longrightarrow \bot}\ \supset L$$

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 $\supset L$.

$$\frac{\neg\neg A \Longrightarrow \neg A \quad \neg\neg A, \bot \Longrightarrow \neg A}{\neg\neg A \Longrightarrow \neg A}\ \supset L$$

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 Identity

We permit the init rule only for atomic propositions. However, the version of this rule with 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 7 (Identity) *For any proposition A, we have $A \Longrightarrow A$.*

Proof: By induction on the structure of A. We show several representative cases and leave the remaining ones to the reader.

Case: $A = P$. Then

$$\frac{}{P \Longrightarrow P}\ \mathsf{init}$$

Case: $A = A_1 \wedge A_2$. Then

$$\dfrac{\dfrac{\dfrac{\text{By i.h. on } A_1 \text{ and weakening}}{A_1 \wedge A_2, A_1 \Longrightarrow A_1}}{A_1 \wedge A_2 \Longrightarrow A_1}\ \wedge L_1 \qquad \dfrac{\dfrac{\text{By i.h. on } A_2 \text{ and weakening}}{A_1 \wedge A_2, A_2 \Longrightarrow A_2}}{A_1 \wedge A_2 \Longrightarrow A_2}\ \wedge L_2}{A_1 \wedge A_2 \Longrightarrow A_1 \wedge A_2}\ \wedge R$$

Case: $A = A_1 \supset A_2$. Then

$$\dfrac{\dfrac{\text{By i.h. on } A_1 \text{ and weakening}}{A_1 \supset A_2, A_1 \Longrightarrow A_1} \qquad \dfrac{\text{By i.h. on } A_2 \text{ and weakening}}{A_1 \supset A_2, A_1, A_2 \Longrightarrow A_2}\ \supset L}{A_1 \supset A_2, A_1 \Longrightarrow A_2}\ \supset R$$

$$A_1 \supset A_2 \Longrightarrow A_1 \supset A_2$$

Case: $A = \bot$. Then

$$\frac{}{\bot \Longrightarrow \bot}\ \bot L$$

□

The identity theorem is the global version of the local completeness property for each individual connective. One can recognize the local expansion as embodied in each case of the inductive proof of identity.

In the next lecture we will see a new theorem, called *cut*, which is the global analogue of local soundness.

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
September 29, 2009

1 Introduction

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

Because it is very easy to go back and forth between sequent calculus deductions of $A\ \mathit{right}$ 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 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* $\Gamma \Longrightarrow A$ *and* $\Gamma, A \Longrightarrow C$ *then* $\Gamma \Longrightarrow C$.

Proof: By nested inductions on the structure of A, the derivation $\mathcal{D}$ of $\Gamma \Longrightarrow A$ and $\mathcal{E}$ of $\Gamma, A \Longrightarrow C$. 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

$$\overset{\mathcal{D}}{\Gamma \Longrightarrow A} \qquad \text{and} \qquad \overset{\mathcal{E}}{\Gamma, A \Longrightarrow C} \qquad \text{to} \qquad \overset{\mathcal{F}}{\Gamma \Longrightarrow C}$$

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 $\Gamma \Longrightarrow C$ from the two given derivations is naturally non-deterministic.

Case: $\mathcal{D}$ is an initial sequent.

$$\mathcal{D} = \frac{}{\Gamma', P \Longrightarrow P}\ \mathsf{init}$$

$\Gamma = (\Gamma', P)$	This case
$\Gamma', P, P \Longrightarrow C$	Deduction $\mathcal{E}$
$\Gamma', P \Longrightarrow C$	By Contraction (see Lecture 9)
$\Gamma \Longrightarrow C$	By equality

Case: $\mathcal{E}$ is an initial sequent using the cut formula.

$$\mathcal{E} = \frac{}{\Gamma, P \Longrightarrow P}\ \mathsf{init}$$

$A = P = C$	This case
$\Gamma \Longrightarrow A$	Deduction $\mathcal{D}$

Case: $\mathcal{E}$ is an initial sequent not using the cut formula.

$$\mathcal{E} = \frac{}{\Gamma', P, A \Longrightarrow P}\ \mathsf{init}$$

$\Gamma = (\Gamma', P)$	This case
$\Gamma', P \Longrightarrow P$	By rule init
$\Gamma \Longrightarrow P$	By equality

Case: A is the principal formula of the final inference in both $\mathcal{D}$ and $\mathcal{E}$. There are a number of subcases to consider, based on the last inference in $\mathcal{D}$ and $\mathcal{E}$. We show some of them.

Subcase:

$$\mathcal{D} = \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \Longrightarrow A_1 & \Gamma \Longrightarrow A_2\end{array}}{\Gamma \Longrightarrow A_1 \wedge A_2}\ \wedge R$$

$$\text{and} \qquad \mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1 \\ \Gamma, A_1 \wedge A_2, A_1 \Longrightarrow C\end{array}}{\Gamma, A_1 \wedge A_2 \Longrightarrow C}\ \wedge L_1$$

$\Gamma, A_1 \Longrightarrow C$	By i.h. on $A_1 \wedge A_2$, $\mathcal{D}$ and $\mathcal{E}_1$
$\Gamma \Longrightarrow C$	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 additionaly 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.

Subcase:

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_2 \\ \Gamma, A_1 \Longrightarrow A_2\end{array}}{\Gamma \Longrightarrow A_1 \supset A_2}\ \supset R$$

$$\text{and} \quad \mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Gamma, A_1 \supset A_2 \Longrightarrow A_1 & \Gamma, A_2 \supset A_2, A_2 \Longrightarrow C\end{array}}{\Gamma, A_1 \supset A_2 \Longrightarrow C}\ \supset L$$

$\Gamma \Longrightarrow A_1$	By i.h. on $A_1 \supset A_2$, $\mathcal{D}$ and $\mathcal{E}_1$
$\Gamma \Longrightarrow A_2$	By i.h. on A_1 from above and $\mathcal{D}_2$
$\Gamma, A_2 \Longrightarrow C$	By i.h. on $A_1 \supset A_2$, $\mathcal{D}$ and $\mathcal{E}_2$
$\Gamma \Longrightarrow C$	By i.h. on A_2 from above

Case: A is not the principal formula of the last inference in $\mathcal{D}$. In that case $\mathcal{D}$ must end in a left rule and we can appeal to the induction hypothesis on one of its premises. We show some of the subcases.

Subcase:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma', B_1 \wedge B_2, B_1 \Longrightarrow A\end{array}}{\Gamma', B_1 \wedge B_2, \Longrightarrow A}\ \wedge L_1$$

$\Gamma = (\Gamma', B_1 \wedge B_2)$	This case
$\Gamma', B_1 \wedge B_2, B_1 \Longrightarrow C$	By i.h. on A, $\mathcal{D}_1$ and $\mathcal{E}$
$\Gamma', B_1 \wedge B_2 \Longrightarrow C$	By rule $\wedge L_1$
$\Gamma \Longrightarrow C$	By equality

Subcase:

$$\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma', B_1 \supset B_2 \Longrightarrow B_1 & \Gamma', B_1 \supset B_2, B_2 \Longrightarrow A\end{array}}{\Gamma', B_1 \supset B_2 \Longrightarrow A}\ \supset L$$

$\Gamma = (\Gamma', B_1 \supset B_2)$	This case
$\Gamma', B_1 \supset B_2, B_2 \Longrightarrow C$	By i.h. on A, $\mathcal{D}_2$ and $\mathcal{E}$
$\Gamma', B_2 \supset B_2 \Longrightarrow C$	By rule $\supset L$ on $\mathcal{D}_1$ and above
$\Gamma \Longrightarrow C$	By equality

Case: A is not the principal formula of the last inference in $\mathcal{E}$. This overlaps with the previous case, 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. We show some of the subcases.

Subcase:

$$\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Gamma, A \Longrightarrow C_1 & \Gamma, A \Longrightarrow C_2\end{array}}{\Gamma, A \Longrightarrow C_1 \wedge C_2}\ \wedge R$$

$C = C_1 \wedge C_2$	This case
$\Gamma \Longrightarrow C_1$	By i.h. on A, $\mathcal{D}$ and $\mathcal{E}_1$
$\Gamma \Longrightarrow C_2$	By i.h. on A, $\mathcal{D}$ and $\mathcal{E}_2$
$\Gamma \Longrightarrow C_1 \wedge C_2$	By rule $\wedge R$ on above

Subcase:

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Gamma', B_1 \wedge B_2, B_1, A \Longrightarrow C\end{array}}{\Gamma', B_1 \wedge B_2, A \Longrightarrow C}\ \wedge L_1$$

$\Gamma = (\Gamma', B_1 \wedge B_2)$	This case
$\Gamma', B_1 \wedge B_2, B_1 \Longrightarrow C$	By i.h. on A, $\mathcal{D}$ and $\mathcal{E}_1$
$\Gamma', B_1 \wedge B_2 \Longrightarrow C$	By rule $\wedge L_1$ from above

□

3 Cut Elimination

Gentzen's original presentation of the sequent calculus included an inference rule for cut. The analogue in our system would be

$$\dfrac{\Gamma \Longrightarrow A \quad \Gamma, A \Longrightarrow C}{\Gamma \Longrightarrow C}\ \mathsf{cut}$$

The advantage of this calculus is that it more directly corresponds to natural deduction in its full generality, rather than verifications. 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 a 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 2 (Cut Elimination) *If $\mathcal{D}$ is a deduction of $\Gamma \Longrightarrow C$ possibly using the* cut *rule, then there exists a cut-free deduction $\mathcal{D}'$ of $\Gamma \Longrightarrow C$.*

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:

$$\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \Longrightarrow A & \Gamma, A \Longrightarrow C\end{array}}{\Gamma \Longrightarrow C}\ \mathsf{cut}$$

$\Gamma \Longrightarrow A$ without cut	By i.h. on $\mathcal{D}_1$
$\Gamma, A \Longrightarrow C$ without cut	By i.h. on $\mathcal{D}_2$
$\Gamma \Longrightarrow C$	By the Cut Theorem

□

Similarly, Gentzen also allowed initial sequents with a non-atomic principal formula. It is a straightforward exercise to show that any deduction that uses non-atomic initial sequents can be expanded into one that uses only atomic ones.

4 Quantification in Sequent Calculus

In natural deduction, we had two forms of hypotheses: A *true* and $c : \tau$ for parameters c. The latter form was introduced into deductions by the $\forall I$ and $\exists E$ rules. In the sequent calculus we make all assumptions explicit on the left-hand side of sequents. In order to model parameters we therefore need a second kind of judgment on the left that reads $c : \tau$. It is customary to collect all such hypotheses in a different context, denoted Σ for *signature*. A sequent then has the form

$$\underbrace{c_1{:}\tau_1, \ldots, c_m{:}\tau_m}_{\Sigma}\ ;\ \underbrace{A_1\ \mathsf{left}, \ldots, A_n\ \mathsf{left}}_{\Gamma} \Longrightarrow C\ \mathsf{right}$$

We assume that all parameters declared in a signature Σ are distinct. Sometimes this requires us to choose a parameter with a name that has not yet been used. When writing down a sequent $\Sigma; \Gamma \Longrightarrow C$ we presuppose that all parameters in Γ and C are declared in Σ. In the bottom-up construction of a deduction we make sure to maintain this.

The typing judgment for terms, $t : \tau$, can depend on the signature Σ but not on logical assumptions A *left*. We therefore write $\Sigma \vdash t : \tau$ to express that term t has type τ in signature Σ.

In all the propositional rules we have so far, the signature Σ is propagated unchanged from the conclusion of the rule to all premises. In order to derive the rules for the quantifiers, we reexamine verifications for guidance, as we did for the propositional rules in Lecture 9.

Universal quantification. We show the verification on the left and with the corresponding right rule.

$$\dfrac{\begin{array}{c}\overline{c : \tau} \\ \vdots \\ A(c)\uparrow\end{array}}{\forall x{:}\tau.\ A(x)\uparrow}\ \forall I^c \qquad \dfrac{\Sigma, c{:}\tau; \Gamma \Longrightarrow A(c)}{\Sigma; \Gamma \Longrightarrow \forall x.\ A(x)}\ \forall R$$

Our general assumption that the signature declares every parameter at most once means that c cannot occur in Σ already or the rule would not apply. Also note that Σ declares all parameters occurring in Γ, so c cannot occur there, either.

The elimination rule that uses a universally quantified assumption corresponds to a left rule.

$$\dfrac{\forall x{:}\tau.\ A(x)\downarrow \quad t : \tau}{A(t)\downarrow}\ \forall E \qquad \dfrac{\Sigma \vdash t : \tau \quad \Sigma; \Gamma, \forall x{:}\tau.\ A(x), A(t) \Longrightarrow C}{\Sigma; \Gamma, \forall x{:}\tau.\ A(x) \Longrightarrow C}\ \forall L$$

Existential quantification. Again, we derive the sequent calculus rules from the introduction and eliminatinon rules.

$$\dfrac{t : \tau \quad A(t)\uparrow}{\exists x{:}\tau.\ A(x)\uparrow}\ \exists I \qquad \dfrac{\Sigma \vdash t : \tau \quad \Sigma; \Gamma \Longrightarrow A(t)}{\Sigma; \Gamma \Longrightarrow \exists x{:}\tau.\ A(x)}\ \exists R$$

As for disjunction elimination, the natural deduction rule already has somewhat of the flavor of the sequent calculs.

$$\dfrac{\exists x{:}\tau.\ A(x)\downarrow \qquad \begin{array}{c}\overline{c:\tau} \quad \overline{A(c)\downarrow}\ u \\ \vdots \\ C\uparrow\end{array}}{C\uparrow}\ \exists E^{c,u} \qquad\qquad \dfrac{\Sigma, c{:}\tau; \Gamma, \exists x{:}\tau.\ A(x), A(c) \Longrightarrow C}{\Sigma; \Gamma, \exists x{:}\tau.\ A(x) \Longrightarrow C}\ \exists L$$

5 Cut Elimination with Quantification

The proof of the cut theorem extends to the case when we add quantifiers. A crucial property we need is substitution for parameters, which corresponds to a similar substitution principle on natural deductions: If $\Sigma \vdash t : \tau$ and $\Sigma, c : \tau; \Gamma \vdash A$ then $\Sigma; [t/c]\Gamma \vdash [t/c]A$. This is proved by a straightforward induction over the structure of the second deduction, appealing to some elementary properties such as weakening where necessary.

We show only one case of the extended proof of cut, where an existential formula is cut and was just introduced on the right and left, respectively.

Subcase:

$$\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{T} & \mathcal{D}_1 \\ \Sigma \vdash t : \tau & \Sigma; \Gamma \Longrightarrow A_1(t)\end{array}}{\Sigma; \Gamma \Longrightarrow \exists x{:}\tau.\ A_1(x)}\ \exists R$$

$$\text{and} \quad \mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Sigma, c{:}\tau; \Gamma, \exists x{:}\tau.\ A_1(x), A_1(c) \Longrightarrow C\end{array}}{\Sigma; \Gamma, \exists x{:}\tau.\ A_1(x) \Longrightarrow C}\ \exists L$$

$\Sigma; \Gamma, \exists x{:}\tau.\ A_1(x), A_1(t) \Longrightarrow C$	By substitution $[t/c]\mathcal{E}_1$
$\Sigma; \Gamma, A_1(t) \Longrightarrow C$	By i.h. on $\exists x.\ A_1(x)$, $\mathcal{D}$, and $[t/c]\mathcal{E}_1$
$\Sigma; \Gamma \Longrightarrow C$	By i.h. on $A_1(t)$, $\mathcal{D}_1$, and above

The induction requires that $A_1(t)$ is considered smaller than $\exists x.\ A_1(x)$. Formally, this can be justified by counting the number of quantifiers and connectives in a proposition and noting that the term t does not contain any. A similar remark applies to check that $[t/c]\mathcal{E}_1$ is smaller than $\mathcal{E}$. Also note how the side condition that c must be a new parameter in the $\exists L$ rule is required in the substitution step to conclude that $[t/c]\Gamma = \Gamma$, $[t/c]A_1(c) = A(t) = [t/x]A_1(x)$, and $[t/c]C = C$.

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 Inversion

15-317: Constructive Logic
Frank Pfenning

Lecture 11
October 6, 2009

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.

2 A More Restrictive Sequent Calculus

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 antecedent. 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 $\Gamma \longrightarrow C$ for a sequent whose deductions try to elimination 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, $\Gamma \Longrightarrow C$.

Conjunction. The right rule works as before; the left rule extracts *both* conjuncts so that the conjunction itself is no longer needed.

$$\frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge R \qquad \frac{\Gamma, A, B \longrightarrow C}{\Gamma, A \wedge B \longrightarrow C}\ \wedge L$$

Observe that for both rules, all premises have smaller sequents than the conclusion if one counts the number of connectives in a 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 $\Gamma \longrightarrow C$ then $\Gamma \Longrightarrow C$. Completeness if generally more difficult. What we want to show is that if $\Gamma \Longrightarrow C$ then also $\Gamma \longrightarrow C$, where the rules for the latter sequents are conceived as more restrictive. The proof of this would 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 completness proof)

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma, A \wedge B, A \Longrightarrow C\end{array}}{\Gamma, A \wedge B \Longrightarrow C}\ \wedge L_1$$

$\Gamma, A \wedge B, A \longrightarrow C$	By i.h. on $\mathcal{D}_1$
$\Gamma, A, B \longrightarrow A$	By identity for $\longrightarrow$
$\Gamma, A \wedge B \longrightarrow A$	By $\wedge L$
$\Gamma, A \wedge B \longrightarrow C$	By cut for $\longrightarrow$

We see that identity and cut for the restricted sequent calculus is needed to show completness in the sense described above. Fortunately, they hold (see further notes at the end of the lecture). We will not formally justify many of the rules, but give informal justifications or counterexamples.

Truth. There is a small surprise here, in that we can have a left rule for $\top$, which eliminates it from the antecedents. It is analogous to the zero-ary case of conjunction.

$$\frac{}{\Gamma \longrightarrow \top}\ \top R \qquad \frac{\Gamma \longrightarrow C}{\Gamma, \top \longrightarrow C}\ \top L$$

Atomic propositions. They are straightforward, since the initial sequents do not change.

$$\frac{}{\Gamma, P \longrightarrow P}\ \mathsf{init}$$

Disjunction. The right rules to do not change; in the left rule we can eliminate the principal formula.

$$\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\ \vee R_1 \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\ \vee R_2 \qquad \frac{\Gamma, A \longrightarrow C \quad \Gamma, B \longrightarrow C}{\Gamma, A \vee B \longrightarrow C}\ \vee L$$

Intuitively, the assumption $A \vee B$ can be eliminated from the ordinary sequent rules because the new assumption A is stronger. More formally:

Case: (of completness proof)

$$\mathcal{D} = \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma, A \vee B, A \Longrightarrow C & \Gamma, A \vee B, B \Longrightarrow C\end{array}}{\Gamma, A \vee B \Longrightarrow C}\ \vee L$$

$\Gamma, A \vee B, A \longrightarrow C$	By i.h. on $\mathcal{D}_1$
$\Gamma, A \longrightarrow A$	By identity for $\longrightarrow$
$\Gamma, A \longrightarrow A \vee B$	By $\vee R_1$
$\Gamma, A \longrightarrow C$	By cut for $\longrightarrow$
$\Gamma, B \longrightarrow B$	By identity for $\longrightarrow$
$\Gamma, B \longrightarrow A \vee B$	By $\vee R_2$
$\Gamma, B \longrightarrow C$	By cut for $\longrightarrow$
$\Gamma, A \vee B \longrightarrow C$	By rule $\vee L$

Falsehood. There is no right rule, and the left rule has no premise to transfers directly.

$$\text{no } \bot R \text{ rule} \qquad \frac{}{\Gamma, \bot \longrightarrow C}\ \bot L$$

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.

$$\frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B}\ \supset R \qquad \frac{\Gamma, A \supset B \longrightarrow A \quad \Gamma, B \longrightarrow C}{\Gamma, A \supset B \longrightarrow C}\ \supset L$$

Here, the assumption $A \supset B$ persists in the first premise but not in the second. Unfortunately, $A \supset B$ may be needed again in that branch of the proof. An example which requires the implication more than once is $\longrightarrow \neg\neg(A \vee \neg A)$, where $\neg A = A \supset \bot$ as usual.

At this point, 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 this in the next lecture.

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 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. But if we apply contraction (which is admissible in the restricted sequent calculus) 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 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 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, consider $\wedge R$. For this to be invertible means that if the conclusion holds than both premises hold. In other words, we have to show: *If* $\Gamma \longrightarrow A \wedge B$ *then* $\Gamma \longrightarrow A$ *and* $\Gamma \longrightarrow B$, which is the opposite of what the rule itself expresses. Fortunately, this follows easily by but, since $\Gamma, A \wedge B \longrightarrow A$ and $\Gamma, A \wedge B \longrightarrow B$.

In order to formalize the strategy of applying inversions eagerly, with-

out 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, Ω is an *ordered context* (say, a stack) that we only access at the right end. Γ^- is a context restricted to those formulas whose left rule are *not* invertible, and C^+ is a formula whose right rule is *not* invertible. Both of these can also contain atoms. After we have developed the rules we will summarize the forms of Γ^- and C^+.

Rather than organizing the presentation by connective, we will follow the judgments, starting on the right.

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.

$$\frac{\Gamma^-; \Omega \xrightarrow{R} A \quad \Gamma^-; \Omega \xrightarrow{R} B}{\Gamma^-; \Omega \xrightarrow{R} A \wedge B} \wedge R \qquad \frac{}{\Gamma^-; \Omega \Longrightarrow \top} \top R \qquad \frac{\Gamma^-; \Omega, A \xrightarrow{R} B}{\Gamma^-; \Omega \xrightarrow{R} A \supset B} \supset R$$

If we encounter an atomic formula, we succeed if it is among the antecedents; otherwise we switch to left inversion.

$$\frac{P \in \Gamma^-}{\Gamma^-; \Omega \xrightarrow{R} P}\ \mathsf{init} \qquad \frac{P \notin \Gamma^- \quad \Gamma^-; \Omega \xrightarrow{L} P}{\Gamma^-; \Omega \xrightarrow{R} P}\ \mathsf{LR}_P$$

If we encounter disjunction or falsehood, we also switch to left inversion.

$$\frac{\Gamma^-; \Omega \xrightarrow{L} A \vee B}{\Gamma^-; \Omega \xrightarrow{R} A \vee B}\ \mathsf{LR}_\vee \qquad \frac{\Gamma^-; \Omega \xrightarrow{L} \bot}{\Gamma^-; \Omega \xrightarrow{R} \bot}\ \mathsf{LR}_\bot$$

Left inversion. The next phase performs left inversion at the right end of the ordered context Ω. Note that for each logical connective or constant,

there is exactly one rule to apply.

$$\frac{\Gamma^-; \Omega, A, B \xrightarrow{L} C^+}{\Gamma^-; \Omega, A \wedge B \xrightarrow{L} C^+} \wedge L \qquad \frac{\Gamma^-; \Omega \xrightarrow{L} C^+}{\Gamma^-; \Omega, \top \xrightarrow{L} C^+} \top L$$

$$\frac{\Gamma^-; \Omega, A \xrightarrow{L} C^+ \quad \Gamma^-; \Omega, B \xrightarrow{L} C^+}{\Gamma^-; \Omega, A \vee B \xrightarrow{L} C^+} \vee L \qquad \frac{}{\Gamma^-; \Omega, \bot \xrightarrow{L} C^+} \bot L$$

For atomic formulas, we look to see if it matches the right-hand side and, if so, succeed. Otherwise, we move it into Γ^-.

$$\frac{P = C^+}{\Gamma^-; \Omega, P \xrightarrow{L} C^+} \text{ init} \qquad \frac{\Gamma^-, P; \Omega \xrightarrow{L} C^+}{\Gamma^-; \Omega, P \xrightarrow{L} C^+} \text{ shift}_P$$

Finally, in the inversion phase, if the formula on the left is an implication, which can not be inverted.

$$\frac{\Gamma^-, A \supset B; \Omega \xrightarrow{L} C^+}{\Gamma^-; \Omega, A \supset B \xrightarrow{L} C^+} \text{ shift}_\supset$$

The search process described so far is deterministic and ether succeeds finitely with a deduction, or we finally have to make a decision we might regret. This is the case when the ordered context has become empty. At this point either one of the $\vee R$ or $\supset L$ rules must be tried.

$$\frac{\Gamma^-; \cdot \xrightarrow{R} A}{\Gamma^-; \cdot \xrightarrow{L} A \vee B} \vee R_1 \qquad \frac{\Gamma^-; \cdot \xrightarrow{R} B}{\Gamma^-; \cdot \xrightarrow{L} A \vee B} \vee R_2 \qquad \frac{\Gamma^-, A \supset B; \cdot \xrightarrow{R} A \quad \Gamma^-; B \xrightarrow{L} C^+}{\Gamma^-, A \supset B; \cdot \xrightarrow{L} C^+} \supset L$$

References

[How98] Jacob M. Howe. *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St. Andrews, Scotland, 1998.

Lecture Notes on Propositional Theorem Proving

15-317: Constructive Logic
Frank Pfenning

Lecture 12
October 8, 2009

1 Introduction

The inversion calculus from the last lecture constitutes a significant step forward, but it still has the problem that in the $\supset L$ rule, the principal formula has to be copied to the first premise. Therefore, the first premise may not be smaller than the conclusion.

We now have two basic choices. One is to refine the idea of loop-checking and make it as efficient as possible. We will not pursue this option here, although it can be done fruitfully [How98, Chapter4].

The second choice is 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. For this purpose we return to the restrictive sequent calculus and postpone for the moment a discussion of inversion. 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.

We repeat the rules of the restrictive sequent calculus here for reference.

$$\frac{}{\Gamma, P \longrightarrow P}\ \mathsf{init}$$

$$\frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge R \qquad \frac{\Gamma, A, B \longrightarrow C}{\Gamma, A \wedge B \longrightarrow C}\ \wedge L$$

$$\frac{}{\Gamma \longrightarrow \top}\ \top R \qquad \frac{\Gamma \longrightarrow C}{\Gamma, \top \longrightarrow C}\ \top L$$

$$\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\ \vee R_1 \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\ \vee R_2 \qquad \frac{\Gamma, A \longrightarrow C \quad \Gamma, B \longrightarrow C}{\Gamma, A \vee B \longrightarrow C}\ \vee L$$

$$\text{no } \bot R \text{ rule} \qquad \frac{}{\Gamma, \bot \longrightarrow C}\ \bot L$$

$$\frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B}\ \supset R \qquad \frac{\Gamma, A \supset B \longrightarrow A \quad \Gamma, B \longrightarrow C}{\Gamma, A \supset B \longrightarrow C}\ \supset L$$

2 Refining the Left Rule for Implication

We consider each possibility for the antecedent of the implication in turn.

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:

$$\frac{\Gamma, B \longrightarrow C}{\Gamma, \top \supset B \longrightarrow C}\ \top\supset L$$

Falsehood. Consider a sequent

$$\Gamma, \bot \supset B \longrightarrow C.$$

Can we find a simpler proposition expressing the same contents? Yes, namely $\top$, since $(\bot \supset B) \equiv \top$. But $\top$ on the left-hand side can be eliminated by $\top L$, so we can specialize the general rule as follows:

$$\frac{\Gamma \longrightarrow C}{\Gamma, \bot \supset B \longrightarrow C}\ \bot\supset L$$

Disjunction. Now we consider a sequent

$$\Gamma, (D \vee E) \supset B \longrightarrow C.$$

Again, we have to ask if there is a simpler equivalent formula we can use instead of $(C \vee D) \supset B$. If we consider the $\vee L$ rule, we might consider $(D \supset B) \wedge (E \supset B)$. A little side calculation confirms that, indeed,

$$((D \vee E) \supset B) \equiv ((D \supset B) \wedge (E \supset B))$$

We can exploit this, playing through the rules as follows

$$\dfrac{\dfrac{\Gamma, D \supset B, E \supset B \longrightarrow C}{\Gamma, (D \supset B) \wedge (E \supset B) \longrightarrow C}\ \wedge L}{\Gamma, (D \vee E) \supset B \longrightarrow C}\ \mathsf{equiv}$$

This suggests the specialized rule

$$\frac{\Gamma, D \supset B, E \supset B \longrightarrow}{\Gamma, (D \vee E) \supset B \longrightarrow C}\ \vee\supset L$$

The question is whether the premise is really smaller than the conclusion in some well-founded measure. We note that both $D \supset B$ and $E \supset B$ are smaller than the original formula $(D \vee E) \supset B$. 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. So, the premise is indeed smaller in the multiset ordering.

Conjunction. Next we consider

$$\Gamma, (D \wedge E) \supset B \longrightarrow C.$$

In this case we can create an equivalent formula by currying.

$$\frac{\Gamma, D \supset (E \supset B) \longrightarrow C}{\Gamma, (D \wedge E) \supset B \longrightarrow C}\ \wedge\supset L$$

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.

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.

Implication. Last, but not least, we consider the case

$$\Gamma, (D \supset E) \supset B \longrightarrow C.$$

We start by playing through the left rule for this particular case because, as we have already seen, implication on the left does not simplify when interacting with another implication.

$$\frac{\dfrac{\Gamma, (D \supset E) \supset B, D \longrightarrow E}{\Gamma, (D \supset E) \supset B \longrightarrow D \supset E}\ {\supset}R \quad \Gamma, B \longrightarrow C}{\Gamma, (D \supset E) \supset B \longrightarrow C}\ {\supset}L$$

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 $((D \supset E) \supset B) \wedge D$. The conjunction here is a representation of two distinguished formulas in the context. Fortunately, we find

$$((D \supset E) \supset B) \wedge D \equiv (E \supset B) \wedge D$$

which can be checked easily. This leads to the specialized rule

$$\frac{\Gamma, E \supset B, D \longrightarrow E \quad \Gamma, B \longrightarrow C}{\Gamma, (D \supset E) \supset B \longrightarrow C}\ {\supset}{\supset}L$$

This concludes the presentation of the specialized rules. The complete set of rule is summarized in Figure 1.

$$\frac{}{\Gamma, P \longrightarrow P}\ \mathsf{init}$$

$$\frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge R \qquad \frac{\Gamma, A, B \longrightarrow C}{\Gamma, A \wedge B \longrightarrow C}\ \wedge L$$

$$\frac{}{\Gamma \longrightarrow \top}\ \top R \qquad \frac{\Gamma \longrightarrow C}{\Gamma, \top \longrightarrow C}\ \top L$$

$$\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\ \vee R_1 \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\ \vee R_2 \qquad \frac{\Gamma, A \longrightarrow C \quad \Gamma, B \longrightarrow C}{\Gamma, A \vee B \longrightarrow C}\ \vee L$$

$$\text{no } \bot R \text{ rule} \qquad \frac{}{\Gamma, \bot \longrightarrow C}\ \bot L$$

$$\frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B}\ \supset R$$

$$\frac{P \in \Gamma \quad \Gamma, B \longrightarrow C}{\Gamma, P \supset B \longrightarrow C}\ P \supset L$$

$$\frac{\Gamma, D \supset (E \supset B) \longrightarrow C}{\Gamma, (D \wedge E) \supset B \longrightarrow C}\ \wedge \supset L \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma, \top \supset B \longrightarrow C}\ \top \supset L$$

$$\frac{\Gamma, D \supset B, E \supset B \longrightarrow}{\Gamma, (D \vee E) \supset B \longrightarrow C}\ \vee \supset L \qquad \frac{\Gamma \longrightarrow C}{\Gamma, \bot \supset B \longrightarrow C}\ \bot \supset L$$

$$\frac{\Gamma, E \supset B, D \longrightarrow E \quad \Gamma, B \longrightarrow C}{\Gamma, (D \supset E) \supset B \longrightarrow C}\ \supset \supset L$$

Figure 1: Contraction-free sequent calculus

3 Asynchronous Decomposition

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 slight 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 question 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 Γ.

Formulas whose left or right rules can always be applied are called left or right *asynchronous*, respectively. We can see by examining the rules and considering the equivalences above and the methods from the last lecture, that the following formulas are asynchrounous:

Right asynchronous	$A \wedge B$, $\top$, $A \supset B$
Left asynchronous	$A \wedge B$, $\top$, $A \vee B$, $\bot$,
	$(D \wedge E) \supset B$, $\top \supset B$, $(D \vee E) \supset B$, $\bot \supset B$

This leaves

Right synchronous	P, $A \vee B$, $\bot$
Left synchronous	P, $P \supset B$, $(D \supset E) \supset B$

Proof search now 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 init or $P{\supset}L$ can be applied and use them if possible. Since these rules are invertible, this does not require a choice. 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 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].

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.

References

[DN00] Roy Dyckhoff and Sara Negri. Admissibility of structural rules for contraction-free systems of intuitionistic logic. *Journal of Symoblic Logic*, 65:1499–1518, 2000.

[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.

[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.

[Sta79] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.

Lecture Notes on Logic Programming

15-317: Constructive Logic
Frank Pfenning

Lecture 13
October 13, 2009

1 Computation vs. Deduction

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 have 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, $15 + 26 \rightarrow (1 + 2 + 1)1 \rightarrow (3 + 1)1 \rightarrow 41$. To *deduce* we start from a conjecture 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, $a^n + b^n \neq c^n$ for $n > 2$, *. . . 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[1] succeeded in reducing a certain class of logical reasoning to computation in so-called Boolean algebras. Since the fundamental undecidability results of the 20th centuries we know that not everything we can reason about is in fact mechanically computable, even if we follow a well-defined set of formal rules.

[1] 1815–1864

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 it establishes theorems. For example, $15 + 26 = 41$ is both the result of a computation, and a theorem of arithmetic. Conversely, deduction can be considered a form of computation if 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.

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.

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 *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 $\cdots$ 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 thruth of propositions for quite some time in this course we generally omit the trailing "*true*". Other examples of judgments on propositions are A *false* (A is false), A *true at* t (A is true at time t, the subject of temporal logic), or K *knows* A (K knows that A is true, the subject of epistemic logic).

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 n and $t_1, \ldots, t_n$ are the arguments. Terms can have variables in them, which we generally denote by upper-case letters. *Atomic propositions* P have the form $p(t_1, \ldots, t_n)$ where p is a *predicate symbol* of arity n and $t_1, \ldots, t_n$ 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 $0, 1, 2, \ldots$ as terms of the form $0, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), \ldots$, using two function symbols (0 of arity 0 and s of arity 1).[2] The first predicate we consider is even of arity 1. Its meaning is defined by two inference rules:

$$\frac{}{\mathsf{even}(0)}\ \mathsf{evz} \qquad\qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\ \mathsf{evs}$$

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 $\mathsf{s}(\mathsf{s}(N))$ 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 defining the predicate even completely.

The following is a trivial example of a deduction, showing that 4 is even:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{even}(0)}\ \mathsf{evz}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))}\ \mathsf{evs}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))))}\ \mathsf{evs}$$

Here, we used the rule evs twice: once with $N = 0$ and once with $N = \mathsf{s}(\mathsf{s}(0))$.

3 Proof Search

To make the transition from inference rules to logic programming we need to impose a particular strategy. Two fundamental ideas suggest themselves: we could either search backward from the conjecture, growing a (potential) proof tree upwards, or we could work forwards from the axioms applying rules until we arrive at the conjecture. We call the first one

[2] This is not how numbers are represented in practical logic programming languages such as Prolog, but it is a convenient source of examples.

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.

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. We will consider many refinements and more precise descriptions of search in this course.

For example, consider the conjecture $\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))$. We now execute the logic program consisting of the two rules evz and evs to either prove or refute this goal. We notice that the only rule with a matching conclusion is evs. Our partial proof now looks like

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{even}(0)\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))}\ \mathsf{evs}$$

with $\mathsf{even}(0)$ as the only subgoal.

Considering the subgoal $\mathsf{even}(0)$ we see that this time only the rule evz could have this conclusion. Moreover, this rule has no premisses so the computation terminates successfully, having found the proof

$$\dfrac{\dfrac{}{\mathsf{even}(0)}\ \mathsf{evz}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))}\ \mathsf{evs}.$$

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.

Now consider the goal $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$. Clearly, since 3 is not even, the computation must fail to produce a proof. Following our strategy, we first reduce this goal using the evs rule to the subgoal $\mathsf{even}(\mathsf{s}(0))$, with the in-

complete proof

$$\dfrac{\dfrac{\vdots}{\mathsf{even}(\mathsf{s}(0))}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))))}\ \mathsf{evs}.$$

At this point we note that there is no rule whose conclusion matches the goal $\mathsf{even}(\mathsf{s}(0))$. 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 $\mathsf{even}(\mathsf{s}(0))$ 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. This means that negation does not have first-class status in logic programming.

4 Answer Substitutions

In the first 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. It is also possible that proof search does not terminate. But how can we write logic programs to compute values?

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 $\mathsf{plus}(m, n, p)$ which should hold if $m + n = p$. We use the recurrence

$$\begin{array}{lcl} (m+1)+n & = & (m+n)+1 \\ 0+n & = & n \end{array}$$

as our guide because it counts down the first argument to 0. We obtain

$$\dfrac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps} \qquad\qquad \dfrac{}{\mathsf{plus}(0, N, N)}\ \mathsf{pz}.$$

Now consider a goal of the form $\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)$ where R is an unknown. This represents the question if there exists an R such that the relation $\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)$ holds. Search not only constructs a proof, but also a term t for R such that $\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), t)$ is true.

For the original goal, $\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)$, only the rule ps could apply because of a mismatch between 0 and $\mathsf{s}(0)$ in the first argument to plus in the conclusion. We also see that the R must have the form $\mathsf{s}(P)$ for some P, although we do not know yet what P should be.

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(0, \mathsf{s}(0), P)\end{array}}{\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)}\ \mathsf{ps} \quad \text{with } R = \mathsf{s}(P)$$

For the subgoal only the pz rule applies and we see that P must equal $\mathsf{s}(0)$.

$$\dfrac{\dfrac{}{\mathsf{plus}(0, \mathsf{s}(0), P)}\ \mathsf{pz} \quad \text{with } P = \mathsf{s}(0)}{\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)}\ \mathsf{ps} \quad \text{with } R = \mathsf{s}(P)$$

If we carry out the substitutions we obtain the complete proof

$$\dfrac{\dfrac{}{\mathsf{plus}(0, \mathsf{s}(0), \mathsf{s}(0))}\ \mathsf{pz}}{\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))}\ \mathsf{ps}$$

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 $R = \mathsf{s}(\mathsf{s}(0))$.

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 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 $\mathsf{plus}(M, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))$, 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, $\mathsf{plus}(0, N, N)$, does not match because the second and third argument of the query are different. However, the rule ps applies and we obtain

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(M_1, \mathsf{s}(0), \mathsf{s}(0))\end{array}}{\mathsf{plus}(M, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))}\ \mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

At this point both rules, ps and pz, match. We use the rule ps because it is listed first, leading to

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(M_2, \mathsf{s}(0), 0)\end{array}}{\mathsf{plus}(M_1, \mathsf{s}(0), \mathsf{s}(0))}\ \mathsf{ps} \quad \text{with } M_1 = \mathsf{s}(M_2)}{\mathsf{plus}(M, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))}\ \mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

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.

$$\dfrac{\dfrac{}{\mathsf{plus}(M_1, \mathsf{s}(0), \mathsf{s}(0))}\ \mathsf{pz} \quad \text{with } M_1 = 0}{\mathsf{plus}(M, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))}\ \mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

At this point the proof is complete, with the answer substitution $M = \mathsf{s}(0)$.

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.

6 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, but operationally the behavior of a program can be quite different.

As an example, we define $\mathsf{times}(m, n, p)$ which should hold if $m \times n = p$. We implement the recurrence

$$\begin{array}{lcl} 0 \times n & = & 0 \\ (m+1) \times n & = & (m \times n) + n \end{array}$$

in the form of the following two inference rules.

$$\frac{}{\mathsf{times}(0, N, 0)}\ \mathsf{tz} \qquad \frac{\mathsf{times}(M, N, P) \quad \mathsf{plus}(P, N, Q)}{\mathsf{times}(\mathsf{s}(M), N, Q)}\ \mathsf{ts}$$

As an example we compute $1 \times 2 = Q$. The first step is determined.

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P) & \mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)\end{array}}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}$$

Now if we solve the left subgoal first, there is only one applicable rule which forces $P = 0$

$$\frac{\begin{array}{cc}\dfrac{}{\mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P)}\ \mathsf{ts}\ (P = 0) & \begin{array}{c}\vdots \\ \mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)\end{array}\end{array}}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}$$

Now since $P = 0$, there is only one rule that applies to the second subgoal and we obtain correctly

$$\frac{\begin{array}{cc}\dfrac{}{\mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P)}\ \mathsf{ts}\ (P = 0) & \dfrac{}{\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{pz}\ (Q = \mathsf{s}(\mathsf{s}(0)))\end{array}}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}.$$

On the other hand, if we solve the right subgoal $\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)$ first we have no information on P and Q, so both rules for plus apply. Since ps is given first, the strategy discussed in the previous section means that we try it first, which leads to

$$\frac{\begin{array}{cc}\begin{array}{c}\vdots \\ \mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P)\end{array} & \dfrac{\begin{array}{c}\vdots \\ \mathsf{plus}(P_1, \mathsf{s}(\mathsf{s}(0)), Q_1)\end{array}}{\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ps}\ (P = \mathsf{s}(P_1), Q = \mathsf{s}(Q_1))\end{array}}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}.$$

Again, rules ps and ts are both applicable, with ps listed first, so we con-

tinue:

$$\dfrac{\mathsf{times}(0,\mathsf{s}(\mathsf{s}(0)),P) \qquad \dfrac{\dfrac{\vdots}{\mathsf{plus}(P_2,\mathsf{s}(\mathsf{s}(0)),Q_2)}\ \mathsf{ps}\ (P_1=\mathsf{s}(P_2),Q_1=\mathsf{s}(Q_2))}{\mathsf{plus}(P_1,\mathsf{s}(\mathsf{s}(0)),Q_1)}\ \mathsf{ps}\ (P=\mathsf{s}(P_1),Q=\mathsf{s}(Q_1))}{\mathsf{times}(\mathsf{s}(0),\mathsf{s}(\mathsf{s}(0)),Q)}\ \mathsf{ts}$$

It is easy to see that this will go on indefinitely, and computation will not terminate.

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. As we will see later when we come to logical frameworks, we can integrate even correctness proofs into the same formalism!

7 Prolog Notation

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[3] of the Prolog newsgroup[4] For this course we use GNU Prolog[5] 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

[3] `http://www.cs.kuleuven.ac.be/~remko/prolog/faq/files/faq.html`
[4] `news://comp.lang.prolog/`
[5] `http://gnu-prolog.inria.fr/`

to a textual format. The Prolog notation for a rule

$$\frac{J_1 \ldots J_n}{J}\ R$$

is

$$J \leftarrow J_1, \ldots, J_n.$$

where the name of the rule is omitted and the left-pointing arrow is rendered as ':-' in a plain text file. We read this as

$$J \textit{ if } J_1 \textit{ and } \cdots \textit{ and } J_n.$$

Prolog terminology for an inference rule is a *clause*, where J is the *head* of the clause and $J_1, \ldots, J_n$ 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.

```
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.

8 Unification

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 the rule may contain schematic variables, and the the goal may also contain variables.

As a simple example (which we glossed over before), consider the goal $\mathsf{plus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), R)$ and the clause $\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P)$. We

need to find some way to instantiate M, N, and P in the clause head and R in the goal such that $\mathsf{plus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), R) = \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))$.

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 $M = \mathsf{0}$, $N = \mathsf{s}(\mathsf{0})$, and $R = \mathsf{s}(P)$. P is arbitrary and remains a variable. Applying these equations to the body of the clause we obtain $\mathsf{plus}(\mathsf{0}, \mathsf{s}(\mathsf{0}), P)$ which will be the subgoal with another logic variable, P.

In order to use the other clause for plus to solve this goal we have to solve $\mathsf{plus}(\mathsf{0}, \mathsf{s}(\mathsf{0}), P) = \mathsf{plus}(\mathsf{0}, N, N)$ which sets $N = \mathsf{s}(\mathsf{0})$ and $P = \mathsf{s}(\mathsf{0})$.

This process is called *unification*, and the equations for the variables we generate represent the *unifier*. 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. Another issue is exemplified by the equation $N = \mathsf{s}(\mathsf{s}(N))$ which does not have a solution: the right-hand side will have 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 could come up if we pose the query $\mathsf{plus}(\mathsf{0}, N, \mathsf{s}(N))$: "*Is there an n such that $0 + n = n + 1$.*"

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.

9 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.

10 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.

11 References

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 14
October 15, 2009

In this lecture we introduce some simple data structures such as lists, and simple algorithms on them such as as quicksort or mergesort. We also introduce some first considerations of types and modes for logic programs.

1 Lists

Lists are defined by two constructors: the empty list nil and the constructor cons which takes an element and a list, generating another list. For example, the list a, b, c would be represented as $\mathsf{cons}(a, \mathsf{cons}(b, \mathsf{cons}(c, \mathsf{nil})))$. The official Prolog notation for nil is `[]`, and for $\mathsf{cons}(h, t)$ is `.(`h`, `t`)`, overloading the meaning of the period '`.`' as a terminator for clauses and a binary function symbol. In practice, however, this notation for cons is rarely used. Instead, most Prolog programs use `[`h`|`t`]` for $\mathsf{cons}(h, t)$.

There is also a sequence notation for lists, so that a, b, c can be written as `[a, b, c]`. It could also be written as `[a | [b | [c | []]]]` or `[a, b | [c, []]]`. Note that all of these notations will be parsed into the same internal form, using nil and cons. We generally follow Prolog list notation in these notes.

2 Type Predicates

We now return to the definition of plus from the previous lecture, except that we have reversed the order of the two clauses.

```
plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

In view of the new list constructors for terms, the first clause now looks wrong. For example, with this clause we can prove

```
plus(s(z), [a, b, c], s([a, b, c])).
```

This is absurd: what does it mean to add 1 and a list? What does the term `s([a, b, c])` denote? It is clearly neither a list nor a number.

From the modern programming language perspective the answer is clear: the definition above lacks *types*. Unfortunately, Prolog (and traditional predicate calculus from which it was originally derived) do not distinguish terms of different types. The historical answer for why these languages have just a single type of terms is that types can be defined as unary predicates. While this is true, it does not account for the pragmatic advantage of distinguishing meaningful propositions from those that are not. To illustrate this, the standard means to correct the example above would be to define a predicate nat with the rules

$$\frac{}{\mathsf{nat}(\mathsf{0})}\ \mathsf{nz} \qquad\qquad \frac{\mathsf{nat}(N)}{\mathsf{nat}(\mathsf{s}(N))}\ \mathsf{ns}$$

and modify the base case of the rules for addition

$$\frac{\mathsf{nat}(N)}{\mathsf{plus}(\mathsf{0}, N, N)}\ \mathsf{pz} \qquad\qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

One of the problems is that now, for example, $\mathsf{plus}(\mathsf{0}, \mathsf{nil}, \mathsf{nil})$ is *false*, when it should actually be meaningless. Many problems in debugging Prolog programs can be traced to the fact that propositions that should be meaningless will be interpreted as either true or false instead, incorrectly succeeding or failing. If we transliterate the above into Prolog, we get:

```
nat(z).
nat(s(N)) :- nat(N).

plus(z, N, N) :- nat(N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

No self-respecting Prolog programmer would write the `plus` predicate this way. Instead, he or she would omit the type test in the first clause leading

to the earlier program. The main difference between the two is whether meaningless clauses are false (with the type test) or true (without the type test). One should then annotate the predicate with the intended domain.

```
% plus(m, n, p) iff m + n = p for nat numbers m, n, p.
plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

It would be much preferable from the programmer's standpoint if this informal comment were a formal type declaration, and an illegal invocation of `plus` were a compile-time error rather than leading to silent success or failure. There has been some significant research on types systems and type checking for logic programming languages [5] and we will talk about types more later in this course.

3 List Types

We begin with the type predicates defining lists.

```
list([]).
list([X|Xs]) :- list(Xs).
```

Unlike languages such as ML, there is no test whether the elements of a list all have the same type. We could easily test whether something is a list of natural numbers.

```
natlist([]).
natlist([N|Ns]) :- nat(N), natlist(Ns).
```

The generic test, whether we are presented with a homogeneous list, all of whose elements satisfy some predicate `P`, would be written as:

```
plist(P, []).
plist(P, [X|Xs]) :- P(X), plist(P, Xs).
```

While this is legal in some Prolog implementations, it can not be justified from the underlying logical foundation, because `P` stands for a predicate and is an argument to another predicate, `plist`. This is the realm of higher-order logic, and a proper study of it requires a development of *higher-order logic programming* [3, 4]. In Prolog the goal `P(X)` is a *meta-call*, often written as `call(P(X))`. We will avoid its use, unless we develop higher-order logic programming later in this course.

4 List Membership and Disequality

As a second example, we consider membership of an element in a list.

$$\frac{}{\mathsf{member}(X, \mathsf{cons}(X, Ys))} \qquad \frac{\mathsf{member}(X, Ys)}{\mathsf{member}(X, \mathsf{cons}(Y, Ys))}$$

In Prolog syntax:

```
% member(X, Ys) iff X is a member of list Ys
member(X, [X|Ys]).
member(X, [Y|Ys]) :- member(X, Ys).
```

Note that in the first clause we have omitted the check whether `Ys` is a proper list, making it part of the presupposition that the second argument to `member` is a list.

Already, this very simple predicate has some subtle points. To show the examples, we use the Prolog notation `?-` A`.` for a query A. After presenting the first answer substitution, Prolog interpreters issue a prompt to see if another solution is desired. If the user types '`;`' the interpreter will backtrack to see if another solution can be found. For example, the query

```
?- member(X, [a,b,a,c]).
```

has four solutions, in the order

```
X = a;
X = b;
X = a;
X = c.
```

Perhaps surprisingly, the query

```
?- member(a, [a,b,a,c]).
```

succeeds twice (both with the empty substitution), once for the first occurrence of `a` and once for the second occurrence.

If `member` is part of a larger program, backtracking of a later goal could lead to unexpected surprises when `member` succeeds again. There could also be an efficiency issue. Assume you keep the list in alphabetical order. Then when we find the first matching element there is no need to traverse the remainder of the list, although the `member` predicate above will always do so.

So what do we do if we want to only check membership, or find the first occurrence of an element in a list? Unfortunately, there is no easy answer, because the most straighforward solution

$$\frac{}{\mathsf{member}(X, \mathsf{cons}(X, Ys))} \qquad \frac{X \neq Y \quad \mathsf{member}(X, Ys)}{\mathsf{member}(X, \mathsf{cons}(Y, Ys))}$$

requires disequality which is problematic in the presence of variables. In Prolog notation:

```
member1(X, [X|Ys]).
member1(X, [Y|Ys]) :- X \= Y, member1(X, Ys).
```

When both arguments are ground, this works as expected, giving just one solution to the query

```
?- member1(a, [a,b,a,c]).
```

However, when we ask

```
?- member1(X, [a,b,a,c]).
```

we only get one answer, namely `X = a`. The reason is that when we come to the second clause, we instantiate `Y` to `a` and `Ys` to `[b,a,c]`, and the body of the clause becomes

```
X \= a, member1(X, [b,a,c]).
```

Now we have the problem that we cannot determine if `X` is different from `a`, because `X` is still a variable. Prolog interprets $s \neq t$ as *non-unifiability*, that is, $s \neq t$ succeeds if s and t are not unifiable. But `X` and `a` are unifiable, so the subgoal fails and no further solutions are generated.[1]

There are two attitudes we can take. One is to restrict the use of disequality (and, therefore, here also the use of `member1`) to the case where both sides have no variables in them. In that case disequality can be easily checked without problems. This is the solution adopted by Prolog, and one which we adopt for now.

The second one is to postpone the disequality $s \neq t$ until we can tell from the structure of s and t that they will be different (in which case we

[1] One must remember, however, that in Prolog unification is not sound because it omits the occurs-check, as hinted at in the previous lecture. This also affects the correctness of disequality.

succeed) or the same (in which case the disequality fails). The latter solution requires a much more complicated operational semantics because some goals must be postponed until their arguments become instantiated. This is the general topic of *constructive negation*[2] [1] in the setting of *constraint logic programming* [2, 6].

Disequality is related to the more general question of negation, because $s \neq t$ is the negation of equality, which is a simple predicate that is either primitive, or could be defined with the one clause `X = X`.

5 Simple List Predicates

Now let's explore some other list operations. We start with prefix(xs, ys) which is supposed to hold when the list xs is a prefix of the list ys. The definition is relatively straightforward.

```
prefix([], Ys).
prefix([X|Xs], [X|Ys]) :- prefix(Xs, Ys).
```

Conversely, we can test for a suffix.

```
suffix(Xs, Xs).
suffix(Xs, [Y|Ys]) :- suffix(Xs, Ys).
```

Interestingly, these predicates can be used in a variety of ways. We can check if one list is a prefix of another, we can enumerate prefixes, and we can even enumerate prefixes and lists. For example:

```
?- prefix(Xs,[a,b,c,d]).
Xs = [];
Xs = [a];
Xs = [a,b];
Xs = [a,b,c];
Xs = [a,b,c,d].
```

enumerates all prefixes, while

[2]The use of the word "constructive" here is unrelated to its use in logic.

```
?- prefix(Xs,Ys).
Xs = [];

Xs = [A]
Ys = [A|_];

Xs = [A,B]
Ys = [A,B|_];

Xs = [A,B,C]
Ys = [A,B,C|_];

Xs = [A,B,C,D]
Ys = [A,B,C,D|_];
...
```

enumerates lists together with prefixes. Note that `A`, `B`, `C`, and `D` are variables, as is the underscore _ so that for example `[A|_]` represents any list with at least one element.

A more general prediate is append(xs, ys, zs) which holds when zs is the result of appending xs and ys.

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

append can also be used in different directions, and we can also employ it for alternative definitions of prefix and suffix.

```
prefix2(Xs, Ys) :- append(Xs, _, Ys).
suffix2(Xs, Ys) :- append(_, Xs, Ys).
```

Here we have used anonymous variables '_'. Note that when several underscores appear in a clauses, each one stands for a different anonymous variable. For example, if we want to define a sublist as a suffix of a prefix, we have to name the intermediate variable instead of leaving it anonymous.

```
sublist(Xs, Ys) :- prefix(Ps, Ys), suffix(Xs, Ps).
```

6 Sorting

As a slightly larger example, we use a recursive definition of quicksort. This is particularly instructive as it clarifies the difference between a specification and an implemention. A specification for sort(xs, ys) would simply say that ys is an ordered permutation of xs. However, this specification is not useful as an implementation: we do not want to cycle through all possible permutations until we find one that is ordered.

Instead we implement a non-destructive version of quicksort, modeled after similar implementations in functional programming. We use here the built-in Prolog integers, rather than the unary representation from the previous lecture. Prolog integers can be compared with n `=<` m (n is less or equal to m) and n `>` m (n is greater than m) and similar predicates, written in infix notation. In order for these comparisons to make sense, the arguments must be instantiated to actual integers and are not allowed to be variables, which constitute a run-time error. This combines two conditions: the first, which is called a *mode*, is that =< and < require their arguments to be *ground* upon invocation, that is not contain any variables. The second condition is a type condition which requires the arguments to be integers. Since these conditions cannot be enforced at compile time, they are signaled as run-time errors.

Quicksort proceeds by partitioning the tail of the input list into those elements that are smaller than or equal to its first element and those that are larger than its first element. It then recursively sorts the two sublists and appends the results.

```
quicksort([], []).
quicksort([X0|Xs], Ys) :-
        partition(Xs, X0, Ls, Gs),
        quicksort(Ls, Ys1),
        quicksort(Gs, Ys2),
        append(Ys1, [X0|Ys2], Ys).
```

Partitioning a list about the pivot element `X0` is also straightforward.

```
partition([], _, [], []).
partition([X|Xs], X0, [X|Ls], Gs) :-
        X =< X0, partition(Xs, X0, Ls, Gs).
partition([X|Xs], X0, Ls, [X|Gs]) :-
        X > X0, partition(Xs, X0, Ls, Gs).
```

Note that the second and third case are both guarded by comparisons. This will fail if either `X` or `X0` are uninstantiated or not integers. The predicate `partition(`xs`, `x_0`, `ls`, `gs`)` therefore inherits a mode and type restriction: the first argument must be a ground list of integers and the second argument must be a ground integer. If these conditions are satisfied and `partition` succeeds, the last two arguments will always be lists of ground integers. In a future lecture we will discuss how to enforce conditions of this kind to discover bugs early. Here, the program is small, so we can get by without mode checking and type checking.

It may seem that the check `X > X0` in the last clause is redundant. However, that is not the case because upon backtracking we might select the second clause, even if the first one succeeded earlier, leading to an incorrect result. For example, without this guard the query

```
?- quicksort([2,1,3], Ys)
```

would incorrectly return `Ys = [2,1,3]` as its second solution.

In this particular case, the test is trivial so the overhead is acceptable. Sometimes, however, a clause is guarded by a complicated test which takes a long time to evaluate. In that case, there is no easy way to avoid evaluating it twice, in pure logic programming. Prolog offers several ways to work around this limitation which we discuss in the next section.

7 Conditionals

We use the example of computing the minimum of two numbers as an example analogous to `partition`, but shorter.

```
minimum(X, Y, X) :- X =< Y.
minimum(X, Y, Y) :- X > Y.
```

In order to avoid the second, redundant test we can use Prolog's *conditional* construct, written as

```
A -> B ; C
```

which solves goal A. If A succeeds we commit to the solution, removing all choice points created during the search for a proof of A and then solve B. If A fails we solve C instead. There is also a short form `A -> B` which is equivalent to `A -> B ; fail` where `fail` is a goal that always fails.

Using the conditional, `minimum` can be rewritten more succinctly as

```
minimum(X, Y, Z) :- X =< Y -> Z = X ; Z = Y.
```

The price that we pay here is that we have to leave the realm of pure logic programming.

Because the conditional is so familiar from imperative and functional program, there may be a tendency to overuse the conditional when it can easily be avoided.

8 Cut

The conditional combines two ideas: commiting to all choices so that only the first solution to a goal will be considered, and branching based on that first solution.

A more powerful primitive is *cut*, written as '`!`', which is unrelated to the use of the word "cut" in proof theory. A cut appears in a goal position and commits to all choices that have been made since the clause it appears in has been selected, including the choice of that clause. For example, the following is a correct implementation of `minimum` in Prolog.

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y).
```

The first clause states that if x is less or equal to y then the minimum is equal to x. Moreover, we commit to this clause in the definition of minimum and on backtracking we do not attempt to use the second clause (which would otherwise be incorrect, of course).

If we permit meta-calls in clauses, then we can define the conditional `A -> B ; C` using cut with

```
if_then_else(A, B, C) :- A, !, B.
if_then_else(A, B, C) :- C.
```

The use of cut in the first clause removes all choice points created during the search for a proof of A when it succeeds for the first time, and also commits to the first clause of `if_then_else`. The solution of B will create choice points and backtrack as usual, except when it fails the second clause of `if_then_else` will never be tried.

If A fails before the cut, then the second clause will be tried (we haven't committed to the first one) and C will be executed.

Cuts can be very tricky and are the source of many errors, because their interpretation depends so much on the operational behavior of the program rather than the logical reading of the program. One should resist the

temptation to use cuts excessively to improve the efficiency of the program unless it is truly necessary.

Cuts are generally divided into *green cuts* and *red cuts*. Green cuts are merely for efficiency, to remove redundant choice points, while red cuts change the meaning of the program entirely. Revisiting the earlier code for `minimum` we see that it is a red cut, since the second clause does not make any sense by itself, but only because the the first clause was attempted before. The cut in

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y) :- X > Y.
```

is a green cut: removing the cut does not change the meaning of the program. It still serves some purpose here, however, because it prevents the second comparison to be carried out if the first one succeeds (although it is still performed redundantly if the first one fails).

A common error is exemplified by the following attempt to make the `minimum` predicate more efficient.

```
% THIS IS INCORRECT CODE
minimum(X, Y, X) :- X =< Y, !.
minimum(X, Y, Y).
```

At first this seems completely plausible, but it is nonetheless incorrect. Think about it before you look at the counterexample at the end of these notes—it is quite instructive.

9 Negation as Failure

One particularly interesting use of cut is to implement negation as finite failure. That is, we say that A is false if the goal A fails. Using higher-order techniques and we can implement `\+(A)` with

```
\+(A) :- A, !, fail.
\+(A).
```

The second clause seems completely contrary to the definition of negation, so we have to interpret this program operationally. To solve `\+(`A`)` we first try to solve A. If that fails we go the second clause which always succeeds. This means that if A fails then `\+(`A`)` will succeed without instantiating any variables. If A succeeds then we commit and fail, so the second clause

will never be tried. In this case, too, no variables are instantiated, this time because the goal fails.

One of the significant problem with negation as failure is the treatment of variables in the goal. That is, `\+(`A`)` succeeds if there is no instance of A that is true. On the other hand, it fails if there is an instance of A that succeeds. This means that free variables may not behave as expected. For example, the goal

```
?- \+(X = a).
```

will fail. According the usual interpretation of free variables this would mean that there is no term t such that $t \neq a$ for the constant a. Clearly, this interpretation is incorrect, as, for example,

```
?- \+(b = a).
```

will succeed.

This problem is similar to the issue we identified for disequality. When goals may not be ground, negation as failure should be viewed with distrust and is probably wrong more often than it is right.

There is also the question on how to reason about logic programs containing disequality, negation as failure, or cut. I do not consider this to be a solved research question.

10 Prolog Arithmetic

As mentioned and exploited above, integers are a built-in data type in Prolog with some predefined predicates such as =< or >. You should consult your Prolog manual for other built-in predicates. There are also some built-in operations such as addition, subtraction, multiplication, and division. Generally these operations can be executed using a special goal of the form t `is` e which evaluates the arithmetic expression e and unifies the result with term t. If e cannot be evaluated to a number, a run-time error will result. As an example, here is the definition of the length predicate for Prolog using built-in integers.

```
% length(Xs, N) iff Xs is a list of length N.
length([], 0).
length([X|Xs], N) :- length(Xs, N1), N is N1+1.
```

As is often the case, the left-hand side of the `is` predicate is a variable, the right-hand side an expression.

11 Answer

The problem is that a query such as

```
?- minimum(5,10,10).
```

will succeed because it fails to match the first clause head.

The general rule of thumb is to leave output variables (here: in the third position) unconstrained free variables and unify it with the desired output after the cut. This leads to the earlier version of `minimum` using cut.

12 References

References

[1] D. Chan. Constructive negation based on the complete database. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICSLP'88)*, pages 111–125, Seattle, Washington, September 1988. MIT Press.

[2] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[3] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[4] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.

[5] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[6] Peter J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS'91)*, pages 328–339, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

Lecture Notes on
Bottom-Up Logic Programming

15-317: Constructive Logic
Frank Pfenning

Lecture 21
November 12, 2009

1 Introduction

In this lecture we return to the view that a logic program is defined by a collection of inference rules for atomic propositions. But we now base the operational semantics on reasoning forward from facts, which are initially given as rules with no premisses. Every rule application potentially adds new facts. Whenever no more new facts can be generated we say forward reasoning *saturates* and we can answer questions about truth by examining the saturated database of facts. We illustrate bottom-up logic programming with several programs, including graph reachability, CKY parsing, and liveness analysis.

2 Bottom-Up Inference

We now return the very origins of logic programming as an operational interpretation of inference rules defining atomic predicates. As a reminder, consider the definition of even.

$$\frac{}{\mathsf{even}(\mathsf{0})}\ \mathsf{evz} \qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\ \mathsf{evss}$$

This works very well on queries such as even(s(s(s(s(s(0)))))) (which succeeds) and even(s(s(s(0)))) (which fails). In fact, the operational reading of this program under goal-directed search constitutes a decision procedure for ground queries even(n).

This specification makes little sense under an alternative interpretation where we eagerly apply the inference rules in the forward direction, from the premisses to the conclusion, until no new facts can be deduced. The problem is that we start with $\mathsf{even}(\mathsf{0})$, then obtain $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{0})))$, and so on, but we never terminate.

It would be too early to give up on forward reasoning at this point. As we have seen many times, even in backward reasoning a natural specification of a predicate does not necessarily lead to a reasonable implementation. We can implement a test whether a number is even via reasoning by contradication. We seed our database with the claim that n is not even and derive consequences from that assumption. If we derive a contradictory fact we know thate $\mathsf{even}(n)$ must be true. If not (and our rules are complete), then $\mathsf{even}(n)$ must be false. We write $\mathsf{odd}(n)$ for the proposition that n is not even. Then we obtain the following specification

$$\frac{\mathsf{odd}(\mathsf{s}(\mathsf{s}(N)))}{\mathsf{odd}(N)}$$

to be used for forward reasoning. This single rule obviously saturates because the argument to odd becomes smaller in every rule application.

What is not formally represented in this program is how we initialize our database (we assume $\mathsf{odd}(n)$), and how we interpret the saturated database (we check if $\mathsf{odd}(\mathsf{0})$ was deduced). In a later lecture we will see that it is possible to combine forward and backward reasoning to makes those aspects of an algorithm also part of its implementation.

The strategy of this example, proof by contradiction, does not always work, but there are many cases where it does. One should check if the predicate is decidable as a first test.

We can also try to stick with the original predicate for even, but add another argument which is a bound to guarantee saturation. We count the bound down, two at a time, while the first argument computes even numbers.

$$\frac{\mathsf{even}(N, \mathsf{s}(\mathsf{s}(B)))}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)), B)}\ \mathsf{evss}$$

When trying to check if a number n is even, we seed the database with

$$\mathsf{even}(\mathsf{0}, n)$$

which expresses that 0 is considered even no matter what the bound (n in this case). Operationally, n is the bound, while 0 is the first even number.

After saturation, which is guaranteed to happen after $\lfloor n/2 \rfloor$ steps, we check if some fact of the form

$$\mathsf{even}(n, \_)$$

is the database.

By an invariant of this algorithm, all deduced facts $\mathsf{even}(b, n)$ have the same sum $b + n$, so we actually know that if n is even we must actually have $\mathsf{even}(n, \mathsf{0})$ at the end, and $\mathsf{even}(n-1, \mathsf{s}(\mathsf{0}))$ if n is odd. This can be used to eliminate the *first* argument, arriving at more or less the code for odd shown above, but with a different justification.

3 Graph Reachability

Assuming we have a specification of $\mathsf{edge}(x, y)$ whenever there is an edge from node x to node y, we can specify reachability $\mathsf{path}(x, y)$ with the rules

$$\frac{\mathsf{edge}(X, Y)}{\mathsf{path}(X, Y)} \qquad \frac{\mathsf{edge}(X, Y) \quad \mathsf{path}(Y, Z)}{\mathsf{path}(X, Z)}$$

During bottom-up inference these rules will saturate when they have constructed the transitive closure of the edge relation. During backward reasoning these rules may not terminate (if there are cycles), or be very inefficient (if there are many paths compared to the number of nodes).

In the forward direction the rules will always saturate. We can also give, just from the rules, a complexity analysis of the saturation algorithm.

4 Complexity Analysis

McAllester [McA02] proved a so-called meta-complexity result which allows us to analyze the structure of a bottom-up logic program and obtain a bound for its asymptotic complexity. We do not review the result or its proof in full detail here, but we sketch it so it can be applied to several of the programs we consider here. Briefly, the result states that the complexity of a bottom-up logic program is $O(|R(D)|+|P_R(R(D))|)$, where $R(D)$ is the saturated database (writing here D for the initial database) and $P_R(R(D))$ is the set of prefix firings of rules R in the saturated database.

The number *prefix firings* for a given rule is computed by analyzing the premisses of the rule from left to right, counting in how many ways it could

match facts in the saturated database. Matching an earlier premiss will fix its variables, which restricts the number of possible matches for later premisses.

For example, in the case of the transitive closure program, assume we have e edges and n vertices. Then in the completed database there can be at most n^2 facts $\mathsf{path}(x, y)$, while there are always exactly e facts $\mathsf{edge}(x, y)$. The first rule

$$\frac{\mathsf{edge}(X, Y)}{\mathsf{path}(X, Y)}$$

can therefore always match in e ways in the completed database. We analyze the premisses of the second rule

$$\frac{\mathsf{edge}(X, Y) \quad \mathsf{path}(Y, Z)}{\mathsf{path}(X, Z)}$$

from left to right. First, $\mathsf{edge}(X, Y)$ can match the database in $O(e)$ ways, as before. This match fixes Y, so there are now $O(n)$ ways that the second premiss could match a fact in the saturated database (each vertex is a candidate for Z). This yields $O(e \cdot n)$ possible prefix firings.

The size of the saturated database is $O(e+n^2)$, and the number of prefix firings of the two rules is $O(e + e \cdot n)$. Therefore the overall complexity is $O(e \cdot n + n^2)$. Since there are up to n^2 edges in the graph, we get a less informative bound of $O(n^3)$ expressed entirely in the number of vertices n.

5 CKY Parsing

Another excellent example for bottom-up logic programming and complexity analysis is a CKY parsing algorithm. This algorithm assumes that the grammar is in Chomsky-normal form, where productions all have the form

$$\begin{aligned} x &\Rightarrow y\, z \\ x &\Rightarrow a \end{aligned}$$

where x, y, and z stand for non-terminals and a for terminal symbols. The idea of the algorithm is to use the grammar production rules from right to left to compute which sections of the input string can be parsed as which non-terminals.

We initialize the database with facts $\mathsf{rule}(x, \mathsf{char}(a))$ for every grammar production $x \Rightarrow a$ and $\mathsf{rule}(x, \mathsf{jux}(y, z))$ for every production $x \Rightarrow y\, z$. We

further represent the input string $a_1 \ldots a_n$ by assumptions $\mathsf{string}(i, a_i)$. For simplicity, we represent numbers in unary form.

Our rules will infer propositions $\mathsf{parse}(x, i, j)$ which we will deduce if the substring $a_i \ldots a_j$ can be parsed as an x. Then the program is represented by the following two rules, to be read in the forward direction:

$$\frac{\mathsf{rule}(X, \mathsf{char}(A)) \quad \mathsf{string}(I, A)}{\mathsf{parse}(X, I, I)} \qquad \frac{\mathsf{rule}(X, \mathsf{jux}(Y, Z)) \quad \mathsf{parse}(Y, I, J) \quad \mathsf{parse}(Z, s(J), K)}{\mathsf{parse}(X, I, K)}$$

After saturating the database with these rules we can see if the whole string is in the language generated by the start symbol s by checking if the fact $\mathsf{parse}(s, \mathsf{s}(\mathsf{0}), n)$ is in the database.

Let g be the number of grammar productions and n the length of the input string. In the completed database we have g grammar rules, n facts $\mathsf{string}(i, a)$, and at most $O(g \cdot n^2)$ facts $\mathsf{parse}(x, i, j)$.

Moving on to the rules, in the first rule there are $O(g)$ ways to match the grammar rule (which fixes A) and then n ways to match $\mathsf{string}(I, A)$, so we have $O(g \cdot n)$. The second rule, again we have $O(g)$ ways to match the grammar rule (which fixes X, Y, and Z) and then $O(n^2)$ ways to match $\mathsf{parse}(Y, I, J)$. In the third premiss now only K is unknown, giving us $O(n)$ way to match it, which means $O(g \cdot n^3)$ prefix firings for the second rule.

These considerations give us an overall complexity of $O(g \cdot n^3)$, which is also the traditional complexity bound for CKY parsing.

6 Liveness Analysis

We consider an application of bottom-up logic programming in program analysis. In this example we analyze code in a compiler's intermediate language to find out which variables are live or dead at various points in the program. We say a variable is *live* at a given program point l if its value will be read before it is written when computation reaches l. This information can be used for optimization and register allocation.

Every command in the language is labeled by an address, which we assume to be a natural number. We use l and k for labels and w, x, y, and z for variables, and op for binary operators. In this stripped-down language we have the following kind of instructions. A representation of the instruction as a logical term is given on the right, although we will

continue to use the concrete syntax to make the rules easier to read.

$$\begin{array}{lcll} l & : & x = op(y,z) & \mathsf{inst}(l, \mathsf{assign}(x, op, y, z)) \\ l & : & \mathsf{if}\ x\ \mathsf{goto}\ k & \mathsf{inst}(l, \mathsf{if}(x, k)) \\ l & : & \mathsf{goto}\ k & \mathsf{inst}(l, \mathsf{goto}(k)) \\ l & : & \mathsf{halt} & \mathsf{inst}(l, \mathsf{halt}) \end{array}$$

We use the proposition $x \neq y$ to check if two variables are distinct and write $\mathsf{s}(l)$ for the successor location to l which contains the next instruction to be executed unless the usual control flow is interrupted.

We write $\mathsf{live}(w, l)$ if we have inferred that variable w is live at l. This is an over-approximation in the sense that $\mathsf{live}(w, l)$ indicates that the variable *may* be live at l, although it is not guaranteed to be read before it is written. This means that any variable that is not live at a given program point is definitely *dead*, which is the information we want to exploit for optimization and register allocation.

We begin with the rules for assignment $x = op(y, z)$. The first two rules just note the use of variables as arguments to an operator. The third one propagates liveness information backwards through the assignment operator. This is sound for any variable, but we would like to achieve that x is not seen as live before the instruction $x = op(y, z)$, so we verify that $W \neq X$.

$$\frac{L : X = Op(Y, Z)}{\mathsf{live}(Y, L)} \qquad \frac{L : X = Op(Y, Z)}{\mathsf{live}(Z, L)} \qquad \frac{\begin{array}{l} L : X = Op(Y, Z) \\ \mathsf{live}(W, \mathsf{s}(L)) \\ W \neq X \end{array}}{\mathsf{live}(W, L)}$$

The rules for jumps propagate liveness information backwards. For unconditional jumps we look at the target; for conditional jumps we look both at the target and the next statement, since we don't analyze whether the condition may be true or false.

$$\frac{\begin{array}{l} L : \mathsf{goto}\ K \\ \mathsf{live}(W, K) \end{array}}{\mathsf{live}(W, L)} \qquad \frac{\begin{array}{l} L : \mathsf{if}\ X\ \mathsf{goto}\ K \\ \mathsf{live}(W, K) \end{array}}{\mathsf{live}(W, L)} \qquad \frac{\begin{array}{l} L : \mathsf{if}\ X\ \mathsf{goto}\ K \\ \mathsf{live}(W, \mathsf{s}(L)) \end{array}}{\mathsf{live}(W, L)}$$

Finally, the variable tested in a conditional is live.

$$\frac{L : \mathsf{if}\ X\ \mathsf{goto}\ K}{\mathsf{live}(X, L)}$$

For the complexity analysis, let n be the number of instructions in the program and v be the number of variables. The size of the saturated database is $O(v \cdot n)$, since all derived facts have the form $\mathsf{live}(X, L)$ where X is a variable and L is the label of an instruction. The prefix firings of all 7 rules are similarly bounded by $O(v \cdot n)$: there are n ways to match the first instruction and then at most v ways to match the second premiss (if any). Hence the overall complexity is bounded by $O(v \cdot n)$.

7 Variable Restrictions

Bottom-up logic programming, as considered by McAllester, requires that every variable in the conclusion of a rule also appears in a premiss. This means that every generated fact will be ground. This is important for saturation and complexity analysis because a fact with a free variable could stand for infinitely many instances.

Nonetheless, bottom-up logic programming can be generalized in the presence of free variables, but we will not discuss this further in this course.

8 Historical Notes

The bottom-up interpretation of logic programs goes back to the early days of logic programming. See, for example, the paper by Naughton and Ramakrishnan [NR91].

There are at least three areas were logic programming specification with a bottom-up semantics has found significant applications: deductive databases, decision procedures, and program analysis. Unification, as present in the next lecture, is an example of a decision procedure for unifiability. Liveness analysis is an example of program analysis first formulated in this fashion by McAllester [McA02], who was particularly interested in describing program analysis algorithms at a high level of abstraction so their complexity would be self-evident. This was later refined by Ganzinger and McAllester [GM01, GM02] by allowing deletions in the database. We treat this in a later lecture where we generalize bottom-up inference to linear logic.

9 Exercises

Exercise 1 *Write a bottom-up logic program for addition (*`plus/3`*) on numbers in unary form and then extend it to multiplication (*`times/3`*).*

Exercise 2 *Consider the following variant of graph reachability.*

$$\frac{\mathsf{edge}(X,Y)}{\mathsf{path}(X,Y)} \qquad \frac{\mathsf{path}(X,Y) \quad \mathsf{path}(Y,Z)}{\mathsf{path}(X,Z)}$$

Perform a McAllester-style complexity analysis and compare the infered complexity with the one given in lecture.

Exercise 3 *The set of prefix firings depends on the order of the premisses. Give an example to demonstrate this.*

Exercise 4 *Extend the bottom-up evaluation semantics for λ-terms by adding rules to compute the substitutions $e(v/x)$. You may assume that v is closed, and that the necessary tests on variable names can be performed.*

Exercise 5 *Relate the bottom-up and top-down version of evaluation of λ-terms to each other by an appropriate pair of theorems.*

Exercise 6 *Add pairs to the evaluation semantics, together with first and second projections. A pair should only be a value if both components are values, that is, pairs are eagerly evaluated.*

Exercise 7 *Give an example which shows that saturation of evaluation for λ-terms may fail to terminate.*

References

[GM01] Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In T.Nipkow R.Goré, A.Leitsch, editor, *Proceedings of the First International Joint Conference on ArAutomated Reasoning (IJCAR'01)*, pages 514–528, Siena, Italy, June 2001. Springer-Verlag LNCS 2083.

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In P.Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[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.