Linear Logic

Frank Pfenning
Carnegie Mellon University

Draft of January 26, 2002

Material for the course *Lineare Logik mit Anwendungen*, Technische Hochschule Darmstadt, Sommersemester 1996. Revised for the course *Linear Logic* at Carnegie Mellon University, Spring 1998, Fall 2001. Material for this course is available at
`http://www.cs.cmu.edu/~fp/courses/linear.html`.

Please send comments to `fp@cs.cmu.edu`

This material is in rough draft form and is likely to contain errors. Furthermore, citations are in no way adequate or complete. Please do not cite or distribute this document.

This work was supported by NSF Grants CCR-9303383 and CCR-9619684.

Contents

Chapter 1

Introduction

In mathematics, one sometimes lives under the illusion that there is just one logic that formalizes the correct principles of mathematical reasoning, the so-called *predicate calculus* or *classical first-order logic*. By contrast, in philosophy and computer science, one finds the opposite: there is a vast array of logics for reasoning in a variety of domains. We mention intuitionistic logic, sorted logic, modal logic, description logic, temporal logic, belief logic, dynamic logic, Hoare logic, specification logic, evaluation logic, relevance logic, higher-order logic, non-monotonic logic, bunched logic, non-commutative logic, affine logic, and, yes, linear logic. Many of these come in a variety of flavors.

There are several reasons for these differing views on logic. An important reason is that in mathematics we use logic only in principle, while in computer science we are interested in using logic in practice. For example, we can eliminate sorts from predicate logic by translating them to predicates and relativizing quantifiers. For example, $\forall x{:}s.\ A(x)$ can be reformulated as $\forall x.\ S(x) \supset A(x)$. This means, in principle, we do not have to bother with sorts when studying logic. On the other hand, practical reasoning with formulas after sorts have been eliminated is much more complex than before. An intrinisic property of an object (t has sort s) has now become a proof obligation ($S(t)$ is true). As a result, some theorem provers such as SPASS instead apply essentially the opposite transformation, translating monadic predicates into sorts before or during the theorem proving process.

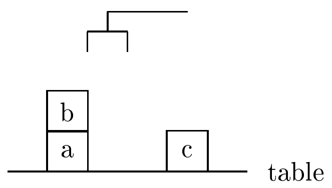
Another important difference between the mathematical and computational point of view lies in the conceptual dependency between the notions of proof and truth. In traditional mathematics we are used to thinking of "truth" as existing abstractly, independently of anyone "knowing" the truth or falsehood of a proposition. Proofs are there to demonstrate truth, but truth is really independent of proof. In computer science, however, we have to be concerned with computation. Proofs in this context show how to construct (= compute) objects whose existence is asserted in a proposition. This means that the notions of construction and proof come before the notion of truth. For example, $\exists x.\ A(x)$ is true if we can construct a t such that $A(t)$ is true. Implication is another example, where $A \supset B$ is true if we can construct a proof of B from a proof of A.

Our approach to linear logic is strongly influenced by both of these points. First, we identify an important problem domain, namely *reasoning with state*, that can be translated into the predicate calculus only with a great deal of coding which makes simple situations appear complex. Second, we develop an appropriate logic *constructively*. This means we explain the meaning of the connectives via their proof rules, and not by an external mathematical semantics. This is both philosophically sound and pragmatically sufficient to understand a logic and how to use it.

Before we launch into examples and informal description of linear logic, we should point out that our perspective is neither historical (linear logic instead arose from domain theory) nor the most popular (much of the current work on linear logic accepts the non-constructive law of excluded middle). On the other hand, we believe our intuitionistic view of linear logic has its own compelling beauty, simplicity, and inevitability, following the tradition of Gentzen [Gen35], Prawitz [Pra65], and Martin-Löf [ML96]. Furthermore, intuitionistic linear logic can directly accomodate most applications that classical linear logic can, but not vice versa.

The interested reader is referred to the original paper by Girard [Gir87], and several surveys [Lin92, Sce93, Tro92] for other views on linear logic. A historical introduction [Doš93] and context for linear and other so-called substructural logics outside computer science can be found in [SHD93].

As a motivating example for linear logic we consider the so-called *blocks world*, which is often used to illustrate planning problems in artificial intelligence. It consists of various blocks stacked on a table and a robot arm that is capable of picking up and putting down one block at a time. We are usually given an initial configuration and some goal to achieve. The diagram below shows typical situation.



We would like to describe this situation, the legal moves, and the problem of achieving a particular goal in logical form. This example led to an independent discovery of a fragment of linear logic by Bibel [Bib86] around the same time that Girard developed linear logic based on a very different foundations.

on(x, y)	block x is on block y
tb(x)	block x is on the table
holds(x)	robot arm holds block x
empty	robot arm is empty
clear(x)	the top of block x is clear

A state is described by a collection of propositions that are true. For example, the state above would be described as

$$\Delta_0 = (\text{empty}, \text{tb}(a), \text{on}(b, a), \text{clear}(b), \text{tb}(c), \text{clear}(c))$$

A goal to be achieved can also be described as a logical proposition such as on(a, b). We would like to develop a logical system so that we can prove a goal G from some assumptions Δ if and only if the goal G can be achieved from the initial state Δ. In this kind of representation, plans correspond to proofs. The immediate problem is how to describe legal moves. Consider the following description:

> *If the robot hand is empty, a block x is clear, and x is on y, then we can pick up the block, that is, achieve a state where the robot hand holds x and y is clear.*

One may be tempted to formulate this as a logical implication.

$$\forall x.\, \forall y.\, (\text{empty} \wedge \text{clear}(x) \wedge \text{on}(x, y)) \supset (\text{holds}(x) \wedge \text{clear}(y))$$

However, this encoding is incorrect. With this axiom we can derive contradictory propositions such as empty$\wedge$holds(b). The problem is clear: logical assumptions persist. In other words, ordinary predicate calculus has no notion of state.

One can try to solve this problem in a number of ways. One way is to introduce a notion of time. If we $\bigcirc A$ to denote the truth of A at the next time, then we might say

$$\forall x.\, \forall y.\, (\text{empty} \wedge \text{clear}(x) \wedge \text{on}(x, y)) \supset \bigcirc(\text{holds}(x) \wedge \text{clear}(y))$$

Now the problem above has been solved, since propositions such as empty $\wedge$ $\bigcirc$holds(b) are not contradictory. However, we now have the opposite problem: we have not expressed that "everything else" stays the same when we pick up a block. Expressing this in temporal logic is possible, but cumbersome. At heart, the problem is that we don't really need a logic of time, but a logic of state.

Miraculously, this is quite easy to achieve by changing our rules on how assumptions may be used. We write

$$A_1\ \mathit{true}, \ldots, A_n\ \mathit{true} \Vdash C\ \mathit{true}$$

to denote that we can prove C from assumptions $A_1, \ldots, A_n$, using every assumption *exactly once*. Another reading of this judgment is:

If we had resources $A_1, \ldots, A_n$ we could achieve goal C.

We refer to the judgment above as a *linear hypothetical judgment*. The order in which assumptions are presented is irrelevant, so we freely allow them to be exchanged. We use the letter Δ to range over a collection of linear assumptions.

From our point of view, the reinterpretation of logical assumptions as consumable resources is the central insight in linear logic from which all else follows in a systematic fashion. Such a seemingly small change has major consequences in properties of the logic and its logical connectives. First, we consider the laws that are derived from the nature of the linear hypothetical judgment itself, without regard to any logical connectives. The first expresses that if we have a resource A we can achieve goal A.

$$\frac{}{A\ \mathit{true} \Vdash A\ \mathit{true}}\ \mathsf{hyp}$$

Note that there may not be any leftover resources, since all resources must be used exactly once. The second law in some sense defines the meaning of linear hypothetical judgments.

If $\Delta \Vdash A$ true and Δ', A true $\Vdash C$ true then $\Delta, \Delta' \Vdash C$ true.

Informally: if we know how to achieve goal A from Δ, and if we know how to achieve C from A and Δ', then we can achieve C if we have both collections of resources, Δ and Δ'. We write Δ, Δ' as concatentation of the resources. This law is called a *substitution principle*, since it allows us to substitute a proof of A *true* for uses of the assumption A *true* in another deduction. The substitution principle does not need to be assumed as a primitive rule of inference. Instead, we want to assure that whenever we can derive the first two judgments, we can already derive the third directly. This expresses that our logical laws have not violated the basic interpretation of the linear hypothetical judgment: we can never obtain more from a resource A than is allowable by our understanding of the linear hypothetical judgment.

Next we introduce a few connectives, considering each in turn.

Simultaneous Conjunction. We write $A \otimes B$ if A and B are true in the same state. For example, we should be able to prove $A\ \mathit{true}, B\ \mathit{true} \Vdash A \otimes B\ \mathit{true}$. The rule for infering a simultaneous conjunction reads

$$\frac{\Delta \Vdash A\ \mathit{true} \qquad \Delta' \Vdash B\ \mathit{true}}{\Delta, \Delta' \Vdash A \otimes B\ \mathit{true}}\ \otimes\mathrm{I}$$

Read from the conclusion to the premises:

In order to achieve goal $A \otimes B$ we divide our resources into Δ and Δ' and show how to achieve A using Δ and B using Δ'.

This is called an *introduction rule*, since it introduce a logical connective in the conclusion. An introduction rule explains the meaning of a connective by explaining how to achieve it as a goal. Conversely, we should also specify how to use our knowledge that we can achieve $A \otimes B$. This is specified in the *elimination rule.*

$$\frac{\Delta \Vdash A \otimes B\ true \qquad \Delta', A\ true, B\ true \Vdash C\ true}{\Delta, \Delta' \Vdash C\ true}\otimes\mathrm{E}$$

We read an elimination rule downward, from the premise to the conclusion:

> *If we know that we can achieve $A \otimes B$ from Δ, we can proceed as if we had both A and B together with some other resources Δ'. Whatever goal C we can achieve form these resources, we can achieve with the joint resources Δ and Δ'.*

Intuitively, it should be clear that this is sound from the meaning of linear hypothetical judgments explained above and summarized in the substitution principle. We will see later more formally how to check that introduction and elimination rules for a connective fit together correctly.

Alternative Conjunction. We write $A \& B$ if we can goals A and B with the current resources, but only *alternatively*. For example, if we have one dollar, we can buy a cup of tea or we can buy a cup of coffee, but we cannot buy them both at the same time. For this reason this is also called *internal choice*. Do not confuse this with disjunction or "exclusive or", the way we often do in natural language! A logical disjunction (also called *external choice*) would correspond to a vending machine that promises to give you tea or coffee, but you cannot choose between them.

The introduction rule for alternative conjunction appears to duplicate the resources.

$$\frac{\Delta \Vdash A\ true \qquad \Delta \Vdash B\ true}{\Delta \Vdash A \& B\ true}\&\mathrm{I}$$

However, this is an illusion: since we will actually have to make a choice between A and B, we will only need one copy of the resources. That we are making an internal choice is also apparent in the elimination rules. If we know how to achieve $A \& B$ we but we have to choose between two rules to obtain either A or B.

$$\frac{\Delta \Vdash A \& B\ true}{\Delta \Vdash A\ true}\&\mathrm{E_L} \qquad \frac{\Delta \Vdash A \& B\ true}{\Delta \Vdash B\ true}\&\mathrm{E_R}$$

Note that we do not use alternative conjunction directly in the blocks world example.

Linear Implication. For our blocks world example, we also need a form of implication: if we had resource A we could achieve B. This is written as $A \multimap B$.

It expresses the meaning of the linear hypothetical judgment as a proposition.

$$\frac{\Delta, A \; true \Vdash B \; true}{\Delta \Vdash A \multimap B \; true} \multimap \mathrm{I}$$

The elimination rule for $A \multimap B$ allows us to conclude that B can be achieved, if we can achieve A.

$$\frac{\Delta \Vdash A \multimap B \; true \qquad \Delta' \Vdash A \; true}{\Delta, \Delta' \Vdash B \; true} \multimap \mathrm{E}$$

Note that we need to join the resources, which should be clear from our intuitive understanding of assumptions as resources.

Without formalizing it, we also assume that we have a universal quantifier with its usual logical meaning. Then we can express the legal moves in the blocks world with the following axioms:

$$\begin{array}{lcl}
\mathrm{geton} & : & \forall x.\, \forall y.\, \mathrm{empty} \otimes \mathrm{clear}(x) \otimes \mathrm{on}(x, y) \multimap \mathrm{holds}(x) \otimes \mathrm{clear}(y), \\
\mathrm{gettb} & : & \forall x.\, \mathrm{empty} \otimes \mathrm{clear}(x) \otimes \mathrm{tb}(x) \multimap \mathrm{holds}(x), \\
\mathrm{puton} & : & \forall x.\, \forall y.\, \mathrm{holds}(x) \otimes \mathrm{clear}(y) \multimap \mathrm{empty} \otimes \mathrm{on}(x, y) \otimes \mathrm{clear}(x), \\
\mathrm{puttb} & : & \forall x.\, \mathrm{holds}(x) \multimap \mathrm{empty} \otimes \mathrm{tb}(x) \otimes \mathrm{clear}(x).
\end{array}$$

Each of these represents a particular possible action, assuming that it can be carried out successfully. Matching the left-hand side of one these rules will consume the corresponding resources so that, for example, the proposition *empty* with no longer be available after the *geton* action has been applied.

For a given state $\Delta = A_1, \ldots, A_n$ we write $\bigotimes \Delta = A_1 \otimes \cdots \otimes A_n$. Then we can reach state Δ' from state Δ if and only if we can prove

$$\Vdash (\bigotimes \Delta) \multimap (\bigotimes \Delta') \; true$$

where the axioms for the legal moves may be used arbitrarily many times. The reader is invited to prove various instances of the planning problem using the rules above.

This is still somewhat unsatisfactory. First of all, we may want to solve a planning problem where not the complete final state, but only some desired aspect of the final state (such as $\mathrm{on}(a, b)$) is specified. Second, the axioms fall outside of the framework on linear hypothetical judgments: they may be used in an unrestricted manner, while state is used linearly.

The first problem is easily remedied by adding another logical constant. The second is more complicated and postponed until the full discussion of natural deduction.

Top. The goal $\top$ can *always* be achieved, regardless of which resources we currently have. We can also think of it as consuming all available resources.

$$\frac{}{\Delta \Vdash \top \; true} \top\mathrm{I}$$

Consequently, we have no information when we know $\top$ and there is no elimination rule. It should be noted that $\top$ is the unit of alternative conjunction in the sense that $A \& \top$ is equivalent to A.

We can use $\top$ in order to specify incomplete goals. For example, if we want to show that we can achieve a state where block a is no b, but we do not care about any other aspect of the state, we can ask if we can prove

$$\Delta_0 \Vdash \mathrm{on}(a,b) \otimes \top$$

where Δ_0 is the representation of the initial state. There is another form of trivial goal we discuss next.

Unit. The goal $\mathbf{1}$ can be achieved if we have no resources.

$$\frac{}{\cdot \Vdash \mathbf{1}\ \mathit{true}}\ \mathbf{1}\mathrm{I}$$

Here we denote the empty collection of resources with "$\cdot$". In this case, knowing $\mathbf{1}$ *true* actually does give us some information, namely that the resources we have can be consumed. This is reflected in the elimination rule.

$$\frac{\Delta \Vdash \mathbf{1}\ \mathit{true} \qquad \Delta' \Vdash C\ \mathit{true}}{\Delta, \Delta' \Vdash C\ \mathit{true}}\ \mathbf{1}\mathrm{E}$$

Multiplicative truth is the unit of $\otimes$ in the sense that $A \otimes \mathbf{1}$ is equivalent to A.

Using our intuitive understanding of the connectives, we can decide various judgments. And, of course, we can back this up with proofs given the rules above. We only give two examples here.

$$A \multimap (B \multimap C)\ \mathit{true} \Vdash (A \otimes B) \multimap C\ \mathit{true}$$

Informally we reason as follows:

In order to show $(A \otimes B) \multimap C$ we assume $A \otimes B$ and show C.
If we know $A \otimes B$ *true* we have both A and B simultaneously.
Using A and $A \multimap (B \multimap C)$ we can then obtain $B \multimap C$.
Using B we can now obtain C.

Note that we use every assumption exactly once in this argument—linearity is preserved.

$$A \otimes B\ \mathit{true} \Vdash A \& B\ \mathit{true}$$

This linear hypothetical judgment cannot be true for arbitrary A and B (although it could be true for some specific A and B). We reason as follows:

Assume $A \otimes B$ *true* $\Vdash A \& B$ *true* holds for arbitrary A and B.
We know A *true*, B *true* $\Vdash A \otimes B$ *true*.

Therefore, by the substitution principle, $A\ \mathit{true}, B\ \mathit{true} \Vdash A \& B\ \mathit{true}$.
We also know $A \& B\ \mathit{true} \Vdash A\ \mathit{true}$ by the hypothesis rule and $\& E_L$.
Therefore $A\ \mathit{true}, B\ \mathit{true} \Vdash A\ \mathit{true}$, again by substitution.
But this is a contradiction to the meaning of the linear hypothetical judgment (B is not used).

Chapter 2

Linear Natural Deduction

Linear logic, in its original formulation by Girard [Gir87] and many subsequent investigations was presented as a refinement of classical logic. This calculus of *classical linear logic* can be cleanly related to classical logic and exhibits many pleasant symmetries. On the other hand, a number of applications in logic and functional programming can be treated most directly using the intuitionistic version. In this chapter we present a basic system of natural deduction defining intuitionistic linear logic.

Our presentation is a judgmental reconstruction of linear logic in the style of Martin-Löf [ML96]. It follows the traditions of Gentzen [Gen35], who first introduced natural deduction, and Prawitz [Pra65], who thoroughly investigated its theory. A similar development of modal logic is given in [PD01]. The way of combining of linear and unrestricted resources goes back to Andreoli [And92] and Girard [Gir93] and, in an explicitly intuitionistic version, Barber [Bar96].

2.1 Judgments and Propositions

In his Siena lectures from 1983 (finally published in 1996), Martin-Löf provides a foundation for logic based on a clear separation of the notions of judgment and proposition. He reasons that to judge is to know and that an evident judgment is an object of knowledge. A proof is what makes a judgment evident. In logic, we make particular judgments such as "A *is a proposition*" or "A *is true*", presupposing in the latter case that A is already known to be a proposition. To know that "A *is a proposition*" means to know what counts as a verification of A, whereas to know that "A *is true*" means to know how to verify A. In his words [ML96, Page 27]:

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

This approach leads to a clear conceptual priority: we first need to understand the notions of judgment and evidence for judgments, then the notions of proposition and verifications of propositions to understand truth.

As an example, we consider the explanation of conjunction. We know that $A \wedge B$ is a proposition if both A and B are propositions. As a rule of inference (called conjunction formation):

$$\frac{A \; prop \qquad B \; prop}{A \wedge B \; prop} \wedge F$$

The meaning is given by stating what counts a verification of $A \wedge B$. We say that we have a verification of $A \wedge B$ if we have verifications for both A and B. As a rule of inference:

$$\frac{A \; true \qquad B \; true}{A \wedge B \; true} \wedge I$$

where we presuppose that A and B are already known to be propositions. This is known as an *introduction rule*, a term due to Gentzen [Gen35] who first formulated a system of natural deduction. Conversely, what do we know if we know that $A \wedge B$ is true? Since a verification of $A \wedge B$ consists of verifications for both A and B, we know that A must be true and B must be true. Formulated as rules of inference (called conjunction eliminations):

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

From the explanation above it should be clear that the two elimination rules are *sound*: if we define the meaning of conjunction by its introduction rule then we are fully justified in concluding that A is true if $A \wedge B$ is true, and similarly for the second rule.

Soundness guarantees that the elimination rules are not too strong. We have sufficient evidence for the judgment in the conclusion if we have sufficient evidence for the judgment in the premise. This is witnessed by a *local reduction* which constructs evidence for the conclusion from evidence for the premise.

$$\frac{\dfrac{\begin{array}{c}\mathcal{D}\\ A \; true\end{array} \qquad \begin{array}{c}\mathcal{E}\\ B \; true\end{array}}{A \wedge B \; true} \wedge I}{A \; true} \wedge E_L \quad \longrightarrow \quad \begin{array}{c}\mathcal{D}\\ A \; true\end{array}$$

A symmetric reduction exists for $\wedge E_R$. We only consider each elimination immediately preceded by an introduction for a connective. We therefore call the property that each such pattern can be reduced *local soundness*.

The dual question, namely if the elimination rules are sufficiently strong, has, as far as we know, not been discussed by Martin-Löf. Of course, we can never achieve "absolute" completeness of rules for inferring evident judgments. But in some situations, elimination rules may be obviously incomplete. For example, we might have overlooked the second elimination rule for conjunction, $\wedge E_R$. This would not contradict soundness, but we would not be able to exploit the knowledge that $A \wedge B$ is true to its fullest. In particular, we cannot recover the knowledge that B is true even if we know that $A \wedge B$ is true.

In general we say that the elimination rules for a connective are *locally complete* if we can apply the elimination rules to a judgment to recover enough knowledge to permit reconstruction of the original judgment. In the case of conjunction, this is only possible if we have both elimination rules.

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

We call this pattern a *local expansion* since we obtain more complex evidence for the original judgment.

An alternative way to understand local completeness is to reconsider our meaning explanation of conjunction. We have said that a verification of $A \wedge B$ consists of a verification of A and a verification of B. Local completeness entails that it is always possible to bring the verification of $A \wedge B$ into this form by a local expansion.

To summarize, logic is based on the notion of judgment where an evident judgment is an object of knowledge. A judgment can be immediately evident or, more typically, mediately evident, in which case the evidence is provided by a proof. The meaning of a proposition is given by what counts as a verification of it. This is written out in the form of introduction rules for logical connectives which allow us to conclude when propositions are true. They are complemented by elimination rules which allow us to obtain further knowledge from the knowledge of compound propositions. The elimination rules for a connective should be locally sound and complete in order to have a satisfactory meaning explanation for the connective. Local soundness and completeness are witnessed by local reductions and expansions of proofs, respectively.

Note that there are other ways to define meaning. For example, we frequently expand our language by *notational definition*. In intuitionistic logic negation is often given as a derived concept, where $\neg A$ is considered a notation for $A \supset \bot$. This means that negation has a rather weak status, as its meaning relies entirely on the meaning of implication and falsehood rather than having an independent explanation. The two should not be mixed: introduction and elimination rules for a connective should rely solely on judgmental concepts and not on other connectives. Sometimes (as in the case of negation) a connective can be explained directly or as a notational definition and we can establish that the two meanings coincide.

2.2 Linear Hypothetical Judgments

So far we have seen two forms of judgment: "*A is a proposition*" and "*A is true*". These are insufficient to explain logical reasoning from assumptions.

For this we need hypothetical judgments and hypothetical proofs, which are new primitive notions. Since we are primarily interested in linear logic, we begin with *linear hypothetical judgments* and *linear hypothetical proofs*. We will postpone discussion of (unrestricted) hypothetical judgments until Section 2.4.

We write the general form of a linear hypothetical judgment as

$$J_1, \ldots, J_n \Vdash J$$

which expresses "*J assuming J_1 through J_n linearly*" or "*J under linear hypotheses J_1 through J_n*". We also refer to $J_1, \ldots, J_n$ as the *antecedents* and J as the *succedent* of the linear hypothetical judgment. The intent of the qualifier "linear" is to indicate that each hypothesis J_i in the antecedent is to be used exactly once. The order of the linear hypotheses is irrelvant, so we will silently allow them to be exchanged.

We now explain what constitutes evidence for a linear hypothetical judgment, namely a linear hypothetical proof. In a hypothetical proof of the judgment above we can use the hypotheses J_i as if they were available as resources. We can consequently substitute an arbitrary derivation of J_i for the uses of a hypothesis J_i to obtain a judgment which no longer depends on J_i. Thus, at the core, the meaning of hypothetical judgments relies upon substitution on the level of proofs, that is, supplanting the use of a hypothesis by evidence for it.

The first particular form of linear hypothetical judgment we need here is

$$u_1{:}A_1\ \mathit{true}, \ldots, u_n{:}A_n\ \mathit{true} \Vdash A\ \mathit{true}$$

where we presuppose that A_1 through A_n and A are all propositions. Note that the propositions A_i do not need to be distinct. We therefore label them with distinct variables u_i so we can refer to them unambiguously. We will sometimes omit the labels for the sake of brevity, but one should keep in mind that

$$A_1\ \mathit{true}, \ldots, A_n\ \mathit{true} \Vdash A\ \mathit{true}$$

is just a shorthand. We write Δ for a collection of linear hypotheses of the form above. The special case of the substitution principle for such hypotheses has the form

Linear Substitution Principle for Truth
If $\Delta \Vdash A\ \mathit{true}$ and $\Delta', u{:}A\ \mathit{true} \Vdash C\ \mathit{true}$ then $\Delta', \Delta \Vdash C\ \mathit{true}$.

Here we write Δ', Δ for the concatenation of two collections of linear hypotheses with distinct labels. We can always rename some labels in Δ or Δ' in order to satisfy this side condition. We further have the general rule for the use of hypotheses.

Linear Hypothesis Rule

$$\frac{}{u{:}A\ \mathit{true} \Vdash A\ \mathit{true}}\ u$$

We sometimes write hyp as the justification for the hypothesis rule if the label u is omitted or irrelevant.

Note that the substitution principle and the linear hypothesis rule together enforce that assumptions are used exactly once. Viewed from the conclusion, the substitution principle splits its resources, distributing it to the two premises. Therefore each assumption in Δ, Δ' will have to be used in either the proof of A or the proof of C from A, but not in both. The linear hypothesis rule does not allow any additional resources among the assumptions besides A, thereby forcing each resource to be used.

We emphasize that the substitution principle should not be viewed as an inference rule, but a property defining hypothetical judgments which we use in the design of a formal system. Therefore it should hold for any system of connectives and inference rules we devise. The correctness of the hypothesis rule, for example, can be seen from the substitution principle.

One further notation: $[\mathcal{D}/u]\mathcal{E}$ is our notation for the result of an appeal to the substitution principle. That is,

$$\text{If} \quad \begin{array}{c}\mathcal{D}\\ \Delta \Vdash A \; \mathit{true}\end{array} \quad \text{and} \quad \begin{array}{c}\mathcal{E}\\ \Delta', u{:}A \Vdash C \; \mathit{true}\end{array} \quad \text{then} \quad \begin{array}{c}[\mathcal{D}/u]\mathcal{E}\\ \Delta, \Delta' \Vdash C \; \mathit{true}\end{array}$$

2.3 Propositions in Linear Logic

Based on the notion of linear hypothetical judgment, we now introduce the various connectives of linear logic via their introduction and elimination rules. We skip, for now, the obvious formation rules for propositions. For each of the connectives we carefully check the local soundness and completeness of the rules and verify the preservation of resources. Also for purely typographical reasons, we abbreviate "A *true*" by just writing "A" in the linear hypothetical judgments.

Simultaneous Conjunction. Assume we have some resources and we want to achieve goals A and B simultaneously, written as $A \otimes B$ (pronounced "A *and* B" or "A *tensor* B"). We need to split our resources into Δ and Δ' and show that with resources Δ we can achieve A and with Δ' we can achieve B.

$$\frac{\Delta \Vdash A \qquad \Delta' \Vdash B}{\Delta, \Delta \Vdash A \otimes B} \otimes I$$

Note that the splitting of resources, viewed bottom-up, is a non-deterministic operation.

The elimination rule should capture what we can achieve if we know that we can achieve both A and B simultaneously from some resources Δ. We reason as follows: If with A, B, and additional resources Δ' we could achieve goal C, then we could achieve C from resources Δ and Δ'.

$$\frac{\Delta \Vdash A \otimes B \qquad \Delta', u{:}A, w{:}B \Vdash C}{\Delta, \Delta' \Vdash C} \otimes E$$

Note that by our general assumption, u and w must be new hypothesis labels in the second premise. The way we achieve C is to commit resources Δ to achieving A and B by the derivation of the left premise and then using the remaining resources Δ' together with A and B to achieve C.

As before, we should check that the rules above are locally sound and complete. First, the local reduction

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Delta_1 \Vdash A\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Delta_2 \Vdash B\end{array}}{\Delta_1, \Delta_2 \Vdash A \otimes B}\otimes\mathrm{I} \qquad \begin{array}{c}\mathcal{E}\\ \Delta', u{:}A, w{:}B \Vdash C\end{array}}{\Delta_1, \Delta_2, \Delta' \Vdash C}\otimes\mathrm{E} \quad \Longrightarrow_R \quad \begin{array}{c}[\mathcal{D}_1/u][\mathcal{D}_2/w]\mathcal{E}\\ \Delta_1, \Delta_2, \Delta' \Vdash C\end{array}$$

which requires two substitutions for linear hypotheses and the application of the substitution principle. The derivation on the right shows that the elimination rules are not too strong.

For local completeness we have the following expansion.

$$\begin{array}{c}\mathcal{D}\\ \Delta \Vdash A \otimes B\end{array} \quad \Longrightarrow_E \quad \dfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash A \otimes B\end{array} \qquad \dfrac{\dfrac{}{u{:}A \Vdash A}u \qquad \dfrac{}{w{:}B \Vdash B}w}{u{:}A, w{:}B \Vdash A \otimes B}\otimes\mathrm{I}}{\Delta \Vdash A \otimes B}\otimes\mathrm{E}$$

The derivation on the right verifies that the elimination rules are strong enough so that the simultaneous conjunction can be reconstituted from the parts we obtain from the elimination rule.

Alternative Conjunction. Next we come to *alternative conjunction* $A \& B$ (pronounced "*A with B*"). It is sometimes also called *internal choice*. In its introduction rule, the resources are made available in both premises, since we have to make a choice which among A and B we want to achieve.

$$\dfrac{\Delta \Vdash A \qquad \Delta \Vdash B}{\Delta \Vdash A \& B}\&\mathrm{I}$$

Consequently, if we have a resource $A \& B$, we can recover either A or B, but not both simultaneously. Therefore we have two elimination rules.

$$\dfrac{\Delta \Vdash A \& B}{\Delta \Vdash A}\&\mathrm{E}_{\mathrm{L}} \qquad \dfrac{\Delta \Vdash A \& B}{\Delta \Vdash B}\&\mathrm{E}_{\mathrm{R}}$$

The local reductions formalize the reasoning above.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash A\end{array} \qquad \begin{array}{c}\mathcal{E}\\ \Delta \Vdash B\end{array}}{\Delta \Vdash A \& B}\&\mathrm{I}}{\Delta \Vdash A}\&\mathrm{E}_{\mathrm{L}} \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{D}\\ \Delta \Vdash A\end{array}$$

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ \Delta \Vdash A\end{matrix} \qquad \begin{matrix}\mathcal{E}\\ \Delta \Vdash B\end{matrix}}{\Delta \Vdash A \& B}\,\&\mathrm{I}}{\Delta \Vdash B}\,\&\mathrm{E_R} \quad \Longrightarrow_R \quad \begin{matrix}\mathcal{E}\\ \Delta \Vdash B\end{matrix}$$

We may recognize these rules from intuitionistic natural deduction, where the assumptions are also available in both premises. The embedding of unrestricted intuitionistic logic in linear logic will therefore map intuitionistic conjunction $A \wedge B$ to alternative conjunction $A \& B$. The expansion is also already familiar.

$$\begin{matrix}\mathcal{D}\\ \Delta \Vdash A \& B\end{matrix} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ \Delta \Vdash A \& B\end{matrix}}{\Delta \Vdash A}\,\&\mathrm{E_L} \qquad \dfrac{\begin{matrix}\mathcal{D}\\ \Delta \Vdash A \& B\end{matrix}}{\Delta \Vdash B}\,\&\mathrm{E_R}}{\Delta \Vdash A \& B}\,\&\mathrm{I}$$

Linear Implication. The *linear implication* or *resource implication* internalizes the linear hypothetical judgment at the level of propositions. We $A \multimap B$ (pronounced "*A linearly implies B*" or "*A lolli B*") for the goal of achieving B with resource A.

$$\dfrac{\Delta, w{:}A \Vdash B}{\Delta \Vdash A \multimap B}\,{\multimap}\mathrm{I}$$

If we know $A \multimap B$ we can obtain B from a derivation of A.

$$\dfrac{\Delta \Vdash A \multimap B \qquad \Delta' \Vdash A}{\Delta, \Delta' \Vdash B}\,{\multimap}\mathrm{E}$$

As in the case for simultaneous conjunction, we have to split the resources, devoting Δ to achieving $A \multimap B$ and Δ' to achieving A.

The local reduction carries out the expected substitution for the linear hypothesis.

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ \Delta, w{:}A \Vdash B\end{matrix}}{\Delta \Vdash A \multimap B}\,{\multimap}\mathrm{I} \qquad \begin{matrix}\mathcal{E}\\ \Delta' \Vdash A\end{matrix}}{\Delta, \Delta' \Vdash B}\,{\multimap}\mathrm{E} \quad \Longrightarrow_R \quad \begin{matrix}[\mathcal{E}/w]\mathcal{D}\\ \Delta, \Delta' \Vdash B\end{matrix}$$

The rules are also locally complete, as witnessed by the local expansion.

$$\begin{matrix}\mathcal{D}\\ \Delta \Vdash A \multimap B\end{matrix} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ \Delta \Vdash A \multimap B\end{matrix} \qquad \dfrac{}{w{:}A \Vdash A}\,w}{\Delta, w{:}A \Vdash B}\,{\multimap}\mathrm{E}}{\Delta \Vdash A \multimap B}\,{\multimap}\mathrm{I}$$

Unit. The trivial goal which requires no resources is written as $\mathbf{1}$.

$$\frac{}{\cdot \Vdash \mathbf{1}}\,\mathbf{1}\mathrm{I}$$

If we can achieve $\mathbf{1}$ from some resources Δ we know that we can consume all those resources.

$$\frac{\Delta \Vdash \mathbf{1} \qquad \Delta' \Vdash C}{\Delta, \Delta' \Vdash C}\,\mathbf{1}\mathrm{E}$$

The rules above and the local reduction and expansion can be seen as a case of 0-ary simultaneous conjunction. In particular, we will see that $\mathbf{1} \otimes A$ is equivalent to A.

$$\frac{\dfrac{}{\cdot \Vdash \mathbf{1}}\,\mathbf{1}\mathrm{I} \qquad \begin{array}{c}\mathcal{E}\\ \Delta' \Vdash C\end{array}}{\Delta' \Vdash C}\,\mathbf{1}\mathrm{E} \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{E}\\ \Delta' \Vdash C\end{array}$$

$$\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \mathbf{1}\end{array} \quad \Longrightarrow_E \quad \frac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \mathbf{1}\end{array} \qquad \dfrac{}{\cdot \Vdash \mathbf{1}}\,\mathbf{1}\mathrm{I}}{\Delta \Vdash \mathbf{1}}\,\mathbf{1}\mathrm{E}$$

Top. There is also a goal which consumes all resources. It is the unit of alternative conjunction and follows the laws of intuitionistic truth.

$$\frac{}{\Delta \Vdash \top}\,\top\mathrm{I}$$

There is no elimination rule for $\top$ and consequently no local reduction (it is trivially locally sound). The local expansion replaces an arbitrary derivation by the introduction rule.

$$\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \top\end{array} \quad \Longrightarrow_E \quad \frac{}{\Delta \Vdash \top}\,\top\mathrm{I}$$

Disjunction. The *disjunction* $A \oplus B$ (also called *external choice*) is characterized by two introduction rules.

$$\frac{\Delta \Vdash A}{\Delta \Vdash A \oplus B}\,\oplus\mathrm{I_L} \qquad \frac{\Delta \Vdash B}{\Delta \Vdash A \oplus B}\,\oplus\mathrm{I_R}$$

As in the case for intuitionistic disjunction, we therefore have to distinguish two cases when we know that we can achieve $A \oplus B$.

$$\frac{\Delta \Vdash A \oplus B \qquad \Delta', u{:}A \Vdash C \qquad \Delta', w{:}B \Vdash C}{\Delta, \Delta' \Vdash C}\,\oplus\mathrm{E}$$

Note that resources Δ' appear in both branches, since only one of those two derivations will actually be used to achieve C, depending on the derivation of $A \oplus B$. This can be seen from the local reductions.

$$\cfrac{\cfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash A\end{array}}{\Delta \Vdash A \oplus B}\oplus\mathrm{I_L} \qquad \begin{array}{c}\mathcal{E}\\ \Delta', u{:}A \Vdash C\end{array} \qquad \begin{array}{c}\mathcal{F}\\ \Delta', w{:}B \Vdash C\end{array}}{\Delta, \Delta' \Vdash C}\oplus\mathrm{E} \quad \Longrightarrow_R \quad \begin{array}{c}[\mathcal{D}/u]\mathcal{E}\\ \Delta, \Delta' \Vdash C\end{array}$$

$$\cfrac{\cfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash B\end{array}}{\Delta \Vdash A \oplus B}\oplus\mathrm{I_L} \qquad \begin{array}{c}\mathcal{E}\\ \Delta', u{:}A \Vdash C\end{array} \qquad \begin{array}{c}\mathcal{F}\\ \Delta', w{:}B \Vdash C\end{array}}{\Delta, \Delta' \Vdash C}\oplus\mathrm{E} \quad \Longrightarrow_R \quad \begin{array}{c}[\mathcal{D}/w]\mathcal{F}\\ \Delta, \Delta' \Vdash C\end{array}$$

The local expansion is straightforward.

$$\begin{array}{c}\mathcal{D}\\ \Delta \Vdash A \oplus B\end{array} \quad \Longrightarrow_E \quad \cfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash A \oplus B\end{array} \qquad \cfrac{\cfrac{}{u{:}A \Vdash A}u}{u{:}A \Vdash A \oplus B}\vee\mathrm{I_L} \qquad \cfrac{\cfrac{}{w{:}B \Vdash B}w}{w{:}B \Vdash A \oplus B}\vee\mathrm{I_R}}{\Delta \Vdash A \oplus B}\vee\mathrm{E}$$

Impossibility. The *impossibility* $\mathbf{0}$ is the case of a disjunction between zero alternatives and the unit of $\oplus$. There is no introduction rule. In the elimination rule we have to consider no branches.

$$\cfrac{\Delta \Vdash \mathbf{0}}{\Delta, \Delta' \Vdash C}\mathbf{0}\mathrm{E}$$

There is no local reduction, since there is no introduction rule. However, as in the case of falsehood in intuitionistic logic, we have a local expansion.

$$\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \mathbf{0}\end{array} \quad \Longrightarrow_E \quad \cfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \mathbf{0}\end{array}}{\Delta \Vdash \mathbf{0}}\mathbf{0}\mathrm{E}$$

Universal Quantification. Quantifiers do not interact much with linearity. We say $\forall x.\ A$ is true if $[a/x]A$ is true for an arbitrary a. This is an example of a *parametric judgment* that we will discuss in more detail in Section **??**.

$$\cfrac{\Delta \Vdash [a/x]A}{\Delta \Vdash \forall x.\ A}\forall\mathrm{I}^a \qquad\qquad \cfrac{\Delta \Vdash \forall x.\ A}{\Delta \Vdash [t/x]A}\forall\mathrm{E}$$

The label a on the introduction rule is a reminder the parameter a must be "new", that is, it may not occur in Δ or $\forall x.\ A$. In other words, the derivation of the premise must *parametric in a*. The local reduction carries out the substitution for the parameter.

$$\cfrac{\cfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash [a/x]A\end{array}}{\Delta \Vdash \forall x.\ A}\,\forall\mathrm{I}^a}{\Delta \Vdash [t/x]A}\,\forall\mathrm{E} \quad \Longrightarrow_R \quad \begin{array}{c}[t/a]\mathcal{D}\\ \Delta \Vdash [t/x]A\end{array}$$

Here, $[t/a]\mathcal{D}$ is our notation for the result of substituting t for the parameter a throughout the deduction $\mathcal{D}$. For this substitution to preserve the conclusion, we must know that a does not already occur in $\forall x.\ A$ or Δ. The local expansion for universal quantification is even simpler.

$$\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \forall x.\ A\end{array} \quad \Longrightarrow_E \quad \cfrac{\cfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \forall x.\ A\end{array}}{\Delta \Vdash [a/x]A}\,\forall\mathrm{E}}{\Delta \Vdash \forall x.\ A}\,\forall\mathrm{I}^a$$

Existential Quantification. Again, this does not interact very much with resources.

$$\cfrac{\Delta \Vdash [t/x]A}{\Delta \Vdash \exists x.\ A}\,\exists\mathrm{I} \qquad \cfrac{\Delta \Vdash \exists x.\ A \qquad \Delta', w{:}[a/x]A \Vdash C}{\Delta, \Delta' \Vdash C}\,\exists\mathrm{E}^a$$

The second premise of the elimination rule must be parametric in a, which is indicated by the superscript a. In the local reduction we will substitute for this parameter.

$$\cfrac{\cfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash [t/x]A\end{array}}{\Delta \Vdash \exists x.\ A}\,\exists\mathrm{I} \qquad \begin{array}{c}\mathcal{E}\\ \Delta', u{:}[a/x]A \Vdash C\end{array}}{\Delta, \Delta' \Vdash C}\,\exists\mathrm{E}^a \quad \Longrightarrow_R \quad \begin{array}{c}[\mathcal{D}/u][t/a]\mathcal{E}\\ \Delta, \Delta' \Vdash C\end{array}$$

The proviso on occurrences of a guarantees that the conclusion and hypotheses of $[t/a]\mathcal{E}$ have the correct form. The local expansion for existential quantification is also similar to the case for disjunction.

$$\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \exists x.\ A\end{array} \quad \Longrightarrow_E \quad \cfrac{\begin{array}{c}\mathcal{D}\\ \Delta \Vdash \exists x.\ A\end{array} \qquad \cfrac{\cfrac{}{u{:}[a/x]A \Vdash [a/x]A}\,u}{u{:}[a/x]A \Vdash \exists x.\ A}\,\exists\mathrm{I}}{\Delta \Vdash \exists x.\ A}\,\exists\mathrm{E}^a$$

This concludes the purely linear operators. Negation and another version of falsehood are postponed to Section **??**, since they may be formally definable, but their interpretation is somewhat questionable in the context we have established so far.

The connectives we have introduced may be classified as to whether the resources are split among the premises or distributed to the premises. Connectives of the former kind are called *multiplicative*, the latter *additive*. For example, we might refer to simultaneous conjunction also as *multiplicative conjunction* and to *alternative conjunction* as *additive conjunction*. When we line up the operators against each other, we notice some gaps. For example, there seems to be only a multiplicative implication, but no additive implication. Dually, there seems to be only an additive disjunction, but no multiplicative disjunction. This is not an accident and is pursued further in Exercise 2.4.

2.4 Unrestricted Hypotheses in Linear Logic

So far, the main judgment permits only linear hypotheses. This means that the logic is too weak to embed ordinary intuitionistic or classical logic, and we have failed so far to design a true extension. In order to accomodate ordinary intuitionistic or classical reasoning, we introduce a new judgment, "*A is valid*", written A *valid*. We say that A is valid if A is true, independently of the any resources. This means we must be able to prove A without any resources. More formally:

> **Validity**
> A *valid* if $\cdot \Vdash A$ *true*.

Note that validity is not a primitive, but a notion derived from truth and linear hypothetical judgments. The judgment $\cdot \Vdash A$ *true* is an example of a *categorical judgment* that asserts independence from hypotheses and also arises in modal logic [PD01]. We can see that, for example, $A \multimap A$ *valid* and $(A \& B) \multimap A$ *valid* for any propositions A and B.

Validity by itself is a completely straightforward judgment. But matters become interesting when we admit *hypotheses* about the validity of propositions. What laws should govern such hypotheses? Let us assume A *valid*, which means that $\cdot \Vdash A$ *true*. First note, that obtaining an instance of A can be achieved without requiring any resources. This means we can generate as many copies of the resource A as we wish, or we may decide not to generate any copies at all. In other words, uses of an assumption A *valid* should be *unrestricted* rather than linear. If we use "$\vdash$" to separate unrestricted hypotheses from a judgment we are trying to deduce, then our main judgment would have the form

$$B_1\ \mathit{valid}, \ldots, B_m\ \mathit{valid} \vdash (A_1\ \mathit{true}, \ldots, A_n\ \mathit{true} \Vdash C\ \mathit{true})$$

which may be read: *under the assumption that $B_1, \ldots, B_m$ are valid and $A_1, \ldots, A_n$ are true we can prove C*. Alternatively, we could say: *with inexhaustible resources $B_1, \ldots, B_m$ and linear resources $A_1, \ldots, A_n$ we can achieve goal C.*

Instead, we will stick with a more customary way of writing this dual hypothetical judgment form by separating the two forms of assumption by a semi-colon ";". As before, we also label assumptions of either kind with distinct variables.

$$(v_1{:}B_1\ \mathit{valid}, \ldots, v_m{:}B_m\ \mathit{valid}); (u_1{:}A_1\ \mathit{true}, \ldots, u_n{:}A_n\ \mathit{valid}) \vdash C\ \mathit{true}$$

It is critical to remember that the first collection of assumptions is unrestricted while the second collection is linear. We abbreviate unrestricted assumptions by Γ and linear assumptions by Δ.

The valid assumptions are independent of the state and can therefore be used freely when proving other valid assumptions. That is,

Validity under Hypotheses
$\Gamma \vdash A\ \mathit{valid}$ if $\Gamma; \cdot \vdash A\ \mathit{true}$.

From this definition we can directly derive a new form of the substitution principle.

Substitution Principle for Validity
If $\Gamma; \cdot \vdash A\ \mathit{true}$ and $(\Gamma, v{:}A\ \mathit{valid}); \Delta \vdash C\ \mathit{true}$ then $\Gamma; \Delta \vdash C\ \mathit{true}$.

Note that the same unrestricted hypotheses Γ appear in the first two jugments, which contrasts with the linear substitution principle where the linear hypotheses are disjoint. This reflects the fact that assumptions in Γ may be used arbitrarily many times in a proof. Note also that the first judgment expresses $\Gamma \vdash A\ \mathit{valid}$, which is necessary so we can substitute for the assumption that $A\ \mathit{valid}$. For a counterexample see Exercise 2.1.

We also have a new hypothesis rule which stems from the definition of validity: if A is valid than it definitely must be true.

Unrestricted Hypothesis Rule

$$\frac{}{(\Gamma, v{:}A\ \mathit{valid}); \cdot \vdash A\ \mathit{true}}\ v$$

Note that there may not be any linear hypotheses (which would be unused), but there may be additional unrestricted hypotheses since they need not be used.

We now restate the original substitution principle and hypothesis rules for our more general judgment. Their form is determined by the unrestricted nature of the validity assumptions. We assume that comma binds more tightly than semi-colon, but may still parenthesize hypotheses to make the judgments more easily readable.

Substitution Principle for Truth
If $\Gamma; \Delta \vdash A\ \mathit{true}$ and $\Gamma; (\Delta', u{:}A\ \mathit{true}) \vdash C\ \mathit{true}$ then $\Gamma; (\Delta', \Delta) \vdash C\ \mathit{true}$.

Hypothesis Rule

$$\frac{}{\Gamma; u{:}A\ \mathit{true} \vdash A\ \mathit{true}}\ u$$

All the rules we presented for pure linear logic so far are extended by adding the unrestricted context to premises and conclusion (see the rule summary on page 24). At this point, for example, we can capture the blocks work example completely inside linear logic. The idea is that the proposition stating the legal moves do not depend on the current state and are therefore given in Γ.

Returning to the blocks world example, a planning problem is now represented as judgment

$$\Gamma_0; \Delta_0 \vdash A_0$$

where Γ_0 represent the rules which describe the legal operations, Δ_0 is the initial state represented as a context of the propositions which are true, and A is the goal to be achieved. For example, the initial state considered earlier would be represented by

$$\Delta_0 = \mathrm{empty}, \mathrm{tb}(a), \mathrm{on}(b, a), \mathrm{clear}(b), \mathrm{tb}(c), \mathrm{clear}(c)$$

where we have omitted labels for the sake of brevity. The rules are represented by unrestricted hypotheses, since they may be used arbitrarily often in the course of solving a problem. We use the following for rules for picking up or putting down an object. We use the convention that simultaneous conjunction $\otimes$ binds more tightly than linear implication $\multimap$.

$$\begin{array}{lll} \Gamma_0 = & & \\ & \mathrm{geton} & : \forall x.\, \forall y.\, \mathrm{empty} \otimes \mathrm{clear}(x) \otimes \mathrm{on}(x, y) \multimap \mathrm{holds}(x) \otimes \mathrm{clear}(y), \\ & \mathrm{gettb} & : \forall x.\, \mathrm{empty} \otimes \mathrm{clear}(x) \otimes \mathrm{tb}(x) \multimap \mathrm{holds}(x), \\ & \mathrm{puton} & : \forall x.\, \forall y.\, \mathrm{holds}(x) \otimes \mathrm{clear}(y) \multimap \mathrm{empty} \otimes \mathrm{on}(x, y) \otimes \mathrm{clear}(x), \\ & \mathrm{puttb} & : \forall x.\, \mathrm{holds}(x) \multimap \mathrm{empty} \otimes \mathrm{tb}(x) \otimes \mathrm{clear}(x). \end{array}$$

Each of these represents a particular possible action, assuming that it can be carried out successfully. Matching the left-hand side of one these rules will consume the corresponding resources so that, for example, the proposition *empty* with no longer be available after the *geton* action has been applied.

The goal that we would like to achieve $\mathrm{on}(a, b)$, for example, is represented with the aid of using $\top$.

$$A_0 = \mathrm{on}(a, b) \otimes \top$$

Any derivation of the judgment

$$\Gamma_0; \Delta_0 \vdash A_0$$

represents a plan for achieving the goal A_0 from the initial situation state Δ_0.

We now go through a derivation of the particular example above, omitting the unrestricted resources Γ_0 which do not change throughout the derivation. Our first goal is to derive

$$\mathrm{empty}, \mathrm{tb}(a), \mathrm{on}(b, a), \mathrm{clear}(b), \mathrm{tb}(c), \mathrm{clear}(c), \mathrm{empty} \vdash \mathrm{on}(a, b) \otimes \top$$

By using $\otimes$I twice we can prove

$$\mathrm{empty}, \mathrm{on}(b,a), \mathrm{clear}(b) \vdash \mathrm{empty} \otimes \mathrm{clear}(b) \otimes \mathrm{on}(b,a)$$

Using the unrestricted hypothesis rule for *geton* followed by $\forall$E twice and $\multimap$E we obtain

$$\mathrm{empty}, \mathrm{clear}(b), \mathrm{on}(b,a) \vdash \mathrm{holds}(b) \otimes \mathrm{clear}(a)$$

Now we use $\otimes$E with the derivation above as our left premise, to prove our overall goal, leaving us with the goal to derive

$$\mathrm{tb}(a), \mathrm{tb}(c), \mathrm{clear}(c), \mathrm{holds}(b), \mathrm{clear}(a) \vdash \mathrm{on}(a,b) \otimes \top$$

as our right premise. Observe how the original resources Δ_0 have been split between the two premises, and the results from the left premise derivation, $\mathrm{holds}(b)$ and $\mathrm{clear}(a)$ have been added to the description of the situation. The new subgoal has exactly the same form as the original goal (in fact, the conclusion has not changed), but applying the unrestricted assumption *geton* has changed our state.

Proceeding in the same manner, using the rule *puttb* next leaves us with the subgoal

$$\mathrm{tb}(a), \mathrm{tb}(c), \mathrm{clear}(c), \mathrm{clear}(a), \mathrm{empty}, \mathrm{clear}(b), \mathrm{tb}(b) \vdash \mathrm{on}(a,b) \otimes \top$$

We now apply *gettb* using a for x and proceeding as above which gives us a derivation of $\mathrm{holds}(a)$. Instead of $\otimes$E, we now use the substitution principle yielding the subgoal

$$\mathrm{tb}(c), \mathrm{clear}(c), \mathrm{clear}(b), \mathrm{tb}(b), \mathrm{holds}(a) \vdash \mathrm{on}(a,b) \otimes \top$$

With same technique, this time using *puton*, we obtain the subgoal

$$\mathrm{tb}(c), \mathrm{clear}(c), \mathrm{tb}(b), \mathrm{empty}, \mathrm{on}(a,b), \mathrm{clear}(a) \vdash \mathrm{on}(a,b) \otimes \top$$

Now we can conclude the derivation with the $\otimes$I rule, distributing resource $\mathrm{on}(a,b)$ to the left premise, which follows immediately as hypothesis, and distributing the remaining resources to the right premise, where $\top$ follows by $\top$I, ignoring all resources.

Note that different derivations of the original judgment represent different sequences of actions (see Exercise 2.5).

Even though it is not necessary in the blocks world example, in order to embed full intuitionistic (or classical) logic into linear logic, we need connectives that allows us to make unrestricted assumptions. We show two operators of this form. The first is *unrestricted implication*, the second a modal operator expressing validity as a proposition.

Unrestricted Implication. The proof of an unrestricted implication $A \supset B$ allows an unrestricted assumption A *valid* while proving that B is true.

$$\frac{(\Gamma, v{:}A); \Delta \vdash B}{\Gamma; \Delta \vdash A \supset B}\supset\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash A \supset B \qquad \Gamma; \cdot \vdash A}{\Gamma; \Delta \vdash B}\supset\mathrm{E}$$

In the elimination we have to be careful to postulate the *validity* of A rather than just its truth, expressed by requiring that there are no linear hypotheses. The local reduction uses the substitution principle for unrestricted hypotheses.

$$\frac{\dfrac{\begin{array}{c}\mathcal{D}\\(\Gamma, v{:}A); \Delta \vdash B\end{array}}{\Gamma; \Delta \vdash A \supset B}\supset\mathrm{I} \qquad \begin{array}{c}\mathcal{E}\\\Gamma; \cdot \vdash A\end{array}}{\Gamma; \Delta \vdash B}\supset\mathrm{E} \quad \Longrightarrow_R \quad \begin{array}{c}[\mathcal{E}/v]\mathcal{D}\\\Gamma; \Delta \vdash B\end{array}$$

In Exercise 2.2 you are asked to show that the rules would be locally unsound (that is, local reduction is not possible), if the second premise in the elimination rule would be allowed to depend on linear hypotheses. The local expansion requires "weakening", that is, adding unused, unrestricted hypotheses.

$$\begin{array}{c}\mathcal{D}\\\Gamma; \Delta \vdash A \supset B\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\(\Gamma, v{:}A); \Delta \vdash A \supset B\end{array} \qquad \dfrac{}{(\Gamma, v{:}A); \cdot \vdash A}v}{(\Gamma, v{:}A); \Delta \vdash B}\supset\mathrm{E}}{\Gamma; \Delta \vdash A \supset B}\supset\mathrm{I}$$

Here, $\mathcal{D}'$ is constructed from $\mathcal{D}$ by adjoining the unused hypothesis u to every judgment, which does not affect the structure of the derivation.

"Of Course" Modality. Girard [Gir87] observed that there is an alternative way to connect unrestricted and linear hypotheses by internalizing the notion of validity via a modal operator $!A$, pronounced "*of course* A" or "*bang* A".

$$\frac{\Gamma; \cdot \vdash A}{\Gamma; \cdot \vdash\, !A}!\mathrm{I}$$

The elimination rule states that if we can derive $\vdash\, !A$ than we are allowed to use A as an unrestricted hypothesis.

$$\frac{\Gamma; \Delta \vdash\, !A \qquad (\Gamma, v{:}A); \Delta' \vdash C}{\Gamma; (\Delta, \Delta') \vdash C}!\mathrm{E}$$

This pair of rules is locally sound and complete via substitution for a valid assumption.

$$\frac{\dfrac{\begin{array}{c}\mathcal{D}\\\Gamma; \cdot \vdash A\end{array}}{\Gamma; \cdot \vdash\, !A}!\mathrm{I} \qquad \begin{array}{c}\mathcal{E}\\(\Gamma, v{:}A); \Delta' \vdash C\end{array}}{\Gamma; \Delta' \vdash C}!\mathrm{E} \quad \Longrightarrow_R \quad \begin{array}{c}[\mathcal{D}/v]\mathcal{E}\\\Gamma; \Delta' \vdash C\end{array}$$

$$\begin{array}{c}\mathcal{D}\\ \Gamma;\Delta\vdash\,!A\end{array} \quad\Longrightarrow_E\quad \dfrac{\begin{array}{c}\mathcal{D}\\ \Gamma;\Delta\vdash\,!A\end{array} \qquad \dfrac{\overline{(\Gamma,v{:}A);\cdot\vdash A}\;v}{(\Gamma,v{:}A);\cdot\vdash\,!A}\;!\mathrm{I}}{\Gamma;\Delta\vdash\,!A}\;!\mathrm{E}$$

Using the *of course* modality, one can define the unrestricted implication $A \supset B$ as $(!A) \multimap B$. It was this observation which gave rise to Girard's development of linear logic. Under this interpretation, the introduction and elimination rules for unrestricted implication are *derived rules of inference* (see Exercise 2.3).

We now summarize the rules of intuitionistic linear logic. A very similar calculus was developed and analyzed in the categorical context by Barber [Bar96]. It differs from more traditional treatments by Abramsky [Abr93], Troelstra [Tro93], Bierman [Bie94] and Albrecht et al. [ABCJ94] in that structural rules remain completely implicit. The logic we consider here comprises the following logical operators.

$$\begin{array}{llll}\text{Propositions} & A & ::= & P & \text{Atoms}\\ & & & \mid A_1 \multimap A_2 \mid A_1 \otimes A_2 \mid \mathbf{1} & \text{Multiplicatives}\\ & & & \qquad\qquad \mid A_1 \& A_2 \mid \top \mid A_1 \oplus A_2 \mid \mathbf{0} & \text{Additives}\\ & & & \mid \forall x.\ A \mid \exists x.\ A & \text{Quantifiers}\\ & & & \mid A \supset B \mid\ !A & \text{Exponentials}\end{array}$$

Recall that the order of both linear and unrestricted hypotheses is irrelevant, and that all hypothesis label in a judgment must be distinct.

Hypotheses.

$$\overline{\Gamma;u{:}A\vdash A}\;u \qquad \overline{(\Gamma,v{:}A);\cdot\vdash A}\;v$$

Multiplicative Connectives.

$$\dfrac{\Gamma;\Delta_1\vdash A \qquad \Gamma;\Delta_2\vdash B}{\Gamma;(\Delta_1,\Delta_2)\vdash A\otimes B}\;\otimes\mathrm{I} \qquad \dfrac{\Gamma;\Delta\vdash A\otimes B \quad \Gamma;(\Delta',u{:}A,w{:}B)\vdash C}{\Gamma;(\Delta,\Delta')\vdash C}\;\otimes\mathrm{E}$$

$$\dfrac{\Gamma;(\Delta,u{:}A)\vdash B}{\Gamma;\Delta\vdash A\multimap B}\;\multimap\mathrm{I} \qquad \dfrac{\Gamma;\Delta\vdash A\multimap B \qquad \Gamma;\Delta'\vdash A}{\Gamma;(\Delta,\Delta')\vdash B}\;\multimap\mathrm{E}$$

$$\dfrac{}{\Gamma;\cdot\vdash\mathbf{1}}\;\mathbf{1}\mathrm{I} \qquad \dfrac{\Gamma;\Delta\vdash\mathbf{1} \qquad \Gamma;\Delta'\vdash C}{\Gamma;(\Delta,\Delta')\vdash C}\;\mathbf{1}\mathrm{E}$$

Additive Connectives.

$$\frac{\Gamma;\Delta \vdash A \qquad \Gamma;\Delta \vdash B}{\Gamma;\Delta \vdash A \& B}\,\&\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash A \& B}{\Gamma;\Delta \vdash A}\,\&\mathrm{E_L} \qquad \frac{\Gamma;\Delta \vdash A \& B}{\Gamma;\Delta \vdash B}\,\&\mathrm{E_R}$$

$$\frac{}{\Gamma;\Delta \vdash \top}\,\top\mathrm{I} \qquad \textit{no } \top \textit{ elimination}$$

$$\frac{\Gamma;\Delta \vdash A}{\Gamma;\Delta \vdash A \oplus B}\,\oplus\mathrm{I_L} \qquad \frac{\Gamma;\Delta \vdash B}{\Gamma;\Delta \vdash A \oplus B}\,\oplus\mathrm{I_R} \qquad \frac{\Gamma;\Delta \vdash A \oplus B \quad \Gamma;(\Delta', u{:}A) \vdash C \quad \Gamma;(\Delta', w{:}B) \vdash C}{\Gamma;(\Delta,\Delta') \vdash C}\,\oplus\mathrm{E}$$

$$\textit{no } \mathbf{0} \textit{ introduction} \qquad \frac{\Gamma;\Delta \vdash \mathbf{0}}{\Gamma;(\Delta,\Delta') \vdash C}\,\mathbf{0}\mathrm{E}$$

Quantifiers.

$$\frac{\Gamma;\Delta \vdash [a/x]A}{\Gamma;\Delta \vdash \forall x.\ A}\,\forall\mathrm{I}^a \qquad \frac{\Gamma;\Delta \vdash \forall x.\ A}{\Gamma;\Delta \vdash [t/x]A}\,\forall\mathrm{E}$$

$$\frac{\Gamma;\Delta \vdash [t/x]A}{\Gamma;\Delta \vdash \exists x.\ A}\,\exists\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash \exists x.\ A \qquad \Gamma;(\Delta', u{:}[a/x]A) \vdash C}{\Gamma;(\Delta,\Delta') \vdash C}\,\exists\mathrm{E}^a$$

Exponentials.

$$\frac{(\Gamma, v{:}A);\Delta \vdash B}{\Gamma;\Delta \vdash A \supset B}\,\supset\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash A \supset B \qquad \Gamma;\cdot \vdash A}{\Gamma;\Delta \vdash B}\,\supset\mathrm{E}$$

$$\frac{\Gamma;\cdot \vdash A}{\Gamma;\cdot \vdash\, !A}\,!\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash\, !A \qquad (\Gamma, v{:}A);\Delta' \vdash C}{\Gamma;(\Delta',\Delta) \vdash C}\,!\mathrm{E}$$

We close this section with another example that exploits the connectives of linear logic. The first example is a *menu* consisting of various courses which can be obtained for 200 French Francs.

Menu A: FF 200	$\mathrm{FF}(200) \multimap$
Onion Soup or *Clear Broth*	$((\mathrm{OS} \& \mathrm{CB})$
Honey-Glazed Duck	$\otimes\, \mathrm{HGD}$
Peas or *Red Cabbage* (according to season)	$\otimes\, (\mathrm{P} \oplus \mathrm{RC})$
New Potatoes	$\otimes\, \mathrm{NP}$
Chocolate Mousse (FF 30 extra)	$\otimes\, ((\mathrm{FF}(30) \multimap \mathrm{CM}) \& \mathbf{1})$
Coffee (unlimited refills)	$\otimes\, \mathrm{C}$ $\otimes\, (!\mathrm{C}))$

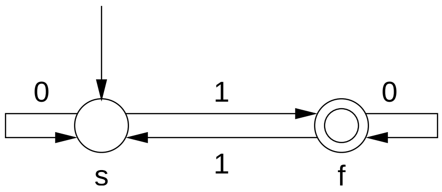
Note the two different informal uses of "or", one modelled by an alternative conjunction and one by a disjunction. The option of ordering chocolate mousse is also represented by an alternative conjunction: we can choose $(\mathrm{FF}(30) \multimap \mathrm{CM}) \& \mathbf{1}$ to obtain nothing ($\mathbf{1}$) or pay another 30 francs to obtain the mousse.

2.5 An Example: Finite Automata

One of the simplest example of computation with state is provided by finite automata. In this section we discuss possible ways to model non-deterministic finite automata in linear logic.

We represent each state of the automaton by a predicate on strings. If the automaton can go from state p to state q while reading string x then we have $p(x) \multimap q(\epsilon)$, where ϵ represents the empty string. More generally, we should have $p(x \cdot y) \multimap q(y)$ for any y, where $x \cdot y$ represent string concatenation. It is convenient to assume we have a single distinguished start state s and final state f. If the automaton has more than one accepting state, we can transform it by adding a new, sole accepting state f and add ϵ-transitions from all the previously accepting states to f.

Consider a simple automaton to accept binary strings with odd parity.



We can implement this with the following propositions, whose use is unrestricted.

$$\begin{array}{rcl} s^0 & : & \forall x.\, s(0 \cdot x) \multimap s(x) \\ s^1 & : & \forall x.\, s(1 \cdot x) \multimap f(x) \\ f^0 & : & \forall x.\, f(0 \cdot x) \multimap f(x) \\ f^1 & : & \forall x.\, f(1 \cdot x) \multimap s(x) \end{array}$$

Even though we do not have the tools to prove this at the moment, we should keep in mind what we would like to achieve. In this example, we can recognize strings with odd parity by adding the unrestricted assumption

$$\forall x.\, (s(x) \multimap f(\epsilon)) \multimap \mathrm{odd}(x).$$

Now we can prove $\mathrm{odd}(x)$ if and only if x is a binary string with odd parity. More generally, our encoding should satisfy the following *adequacy theorem.*

Adequacy for Sequential Encoding of Automata.

> Given a non-deterministic finite automaton M and its encoding Γ. Then for all states p and q and strings x, $p \xrightarrow{x} q$ if and only if $\Gamma; \cdot \vdash \forall y.\, p(x \cdot y) \multimap q(y)$. In particular, if M has initial state s and final state f, then M accepts x if and only if $\Gamma; \cdot \vdash s(x) \multimap f(\epsilon)$.

The direct representation given above maps every possible single-step transition $p \xrightarrow{x} q$ to the proposition $\forall y.\, p(x \cdot y) \multimap q(y)$. Typically, x would be a character c or the empty string ϵ, but we can also translate automata that can accept multiple characters in one step.

Non-deterministic finite automata accept exactly the regular languages as defined by regular expressions. In addition, regular languages are closed under some other operations such as intersection or complement. We now consider how to translate a regular expression into linear logic following a similar strategy as for automata above. In this case we give the construction of the linear logic propositions inductively, based on the shape of the regular expression. We write $\mathcal{L}(r)$ for the language of strings defined by a regular expression.

Adequacy for Sequential Encoding of Regular Expressions.

> Given a regular expression r and its encoding Γ with distinguished predicates s (*start*) and f (*final*). Then $x \in \mathcal{L}(r)$ if and only if $\Gamma; \cdot \vdash \forall y.\, s(x \cdot y) \multimap f(y)$.

For each form of regular expression we now go though the corresponding construction of Γ. We write $\Gamma(s, f)$ to identify the distinguished start and final predicate.

Case: $r = a$ for a character a where $\mathcal{L}(a) = \{a\}$. Then

$$\Gamma(s, f) = \forall y.\, s(a \cdot y) \multimap f(y)$$

Case: $r = r_1 \cdot r_2$ where $\mathcal{L}(r_1 \cdot r_2) = \{x_1 \cdot x_2 \mid x_1 \in \mathcal{L}(r_1) \text{ and } x_2 \in \mathcal{L}(r_2)\}$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translations of r_1 and r_2 respectively. We construct

$$\begin{array}{rcl} \Gamma(s, f) & = & \forall x.\, s(x) \multimap s_1(x), \\ & & \Gamma_1(s_1, f_1), \\ & & \forall z.\, f_1(z) \multimap s_2(z), \\ & & \Gamma_2(s_2, f_2), \\ & & \forall y.\, f_2(y) \multimap f(y) \end{array}$$

Case: $r = \mathbf{1}$ where $\mathcal{L}(\mathbf{1}) = \{\epsilon\}$. Then

$$\Gamma(s, f) = \forall y.\, s(y) \multimap f(y)$$

Case: $r = r_1 + r_2$ where $\mathcal{L}(r_1 + r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translations of r_1 and r_2 respectively. We construct

$$\begin{array}{rcl} \Gamma(s, f) & = & \forall x.\, s(x) \multimap (s_1(x) \,\&\, s_2(x)), \\ & & \Gamma_1(s_1, f_1), \Gamma_2(s_2, f_2), \\ & & \forall y.\, f_1(y) \multimap f(y), \\ & & \forall y.\, f_2(y) \multimap f(y) \end{array}$$

Alternatively, we could replace the first rule with the following two:

$$\begin{array}{l} \forall x.\, s(x) \multimap s_1(x), \\ \forall x.\, s(x) \multimap s_2(x) \end{array}$$

The first formulation may have the slight advantage that every state p except the final state has exactly one transition $\forall x.\ p(t) \multimap A$ for some string t and proposition A. We can also reaplce the last two propositions by

$$\forall y.\ (f_1(y) \oplus f_2(y)) \multimap f(y)$$

This may be preferable if we would like to maintain instead the invariant that every final state has one transition into it. These formulations are equivalent, since

$$\begin{array}{rcl} A \multimap (B \& C) & \dashv\vdash & (A \multimap B) \& (A \multimap C) \\ (A \oplus B) \multimap C & \dashv\vdash & (A \multimap C) \& (B \multimap C) \end{array}$$

and the fact that an *unrestricted* assumption $A \& B$ *valid* is equivalent to two *unrestricted* assumptions A *valid*, B *valid*.

Case: $r = \mathbf{0}$ where $\mathcal{L}(\mathbf{0}) = \{\,\}$.

$$\Gamma(s, f) \quad = \quad \forall x.\ s(x) \multimap \top$$

with no rule for f. In analogy with the previous case, we could also simply not generate any propositions for s and f, or generate the proposition

$$\forall y.\ \mathbf{0} \multimap f(y)$$

for f.

Case: $r = r_1^*$ where $\mathcal{L}(r_1^*) = \{x_1 \cdots x_n \mid x_i \in r_1 \text{ for } 1 \leq i \leq n \text{ and } n \geq 0\}$. Let $\Gamma_1(s_1, f_1)$ be the translation of r_1. Then we construct

$$\begin{array}{rcl} \Gamma(s, f) & = & \forall x.\ s(x) \multimap (f(x) \& s_1(x)), \\ & & \Gamma_1(s_1, f_1), \\ & & \forall y.\ f_1(y) \multimap s(y) \end{array}$$

Alternatively, the first proposition can be broken up into two as in the case for $r_1 + r_2$.

Regular languages are closed under intersection ($\cap$), the full language ($\mathbf{T}$) and complementation. At least the first two are relatively easy to implement.

Case: $r = r_1 \cap r_2$ where $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translation of r_1 and r_2. The difficult part of intersection is that r_1 and r_2 must consume *the same initial segment* of the input. This can be achieved using simultaneous conjunction.

$$\begin{array}{rcl} \Gamma(s, f) & = & \forall x.\ s(x) \multimap (s_1(x) \otimes s_2(x)), \\ & & \Gamma_1(s_1, f_1), \Gamma_2(s_2, f_2), \\ & & \forall y.\ (f_1(y) \otimes f_2(y)) \multimap f(y) \end{array}$$

Note how we exploit multiple linear hypotheses and force the synchronization of their accepting states on y.

Case: $r = \mathbf{T}$ where $\mathcal{L}(\mathbf{T}) = \Sigma^*$, the set of all strings over the alphabet Σ. $\mathbf{T}$ accepts *any* initial segment of the input string.

$$\Gamma(s,f) \quad = \quad \forall x.\, \forall y.\, s(x \cdot y) \multimap f(y)$$

Strictly speaking, this proposition could present some problems in that solving an equation such as $a \cdot b \cdot c = x \cdot y$ has multiple solutions if x and y both stand for arbitrary strings. If we would like to avoid the assumption that the logic understands the equational theory of strings, we could decompose this clause into

$$\begin{array}{rcl}\Gamma(s,f) & = & \forall x.\, s(x) \multimap f(x), \\ & & \forall a.\, \forall y.\, s(a \cdot y) \multimap s(y)\end{array}$$

where a ranges only over characters.

Complement appears to be more complicated, and we presently have no direct and elegant solution. Note that the encoding of $\mathbf{T}$ we take some algebraic properties of string concatenation for granted without axiomatizing them explicitly.

In the representation, non-determinism arising from $r_1 + r_2$ is represented by an internal choice in

$$\forall x.\, s(x) \multimap (s_1(x) \& s_2(x)).$$

That is, in the course of the proof itself we have to guess (internal choice) whether $s_1(x)$ or $s_2(x)$ will lead to success.

An alternative model of computation would try all successor states essentially concurrently. The corresponds to the idea for transforming non-deterministic automata into deterministic ones: we keep track of all the possible states we might be in instead of guessing which transition to make. While in the encoding above, we have essentially one linear hypothesis (the current state, applied to the remaining input string), here we will have multiple ones. The adequacy for this kind of representation is more difficult to formulate precisely, because the additional threads of computation.

Adequacy of Concurrent Encoding of Automata.

> Given a non-deterministic finite automaton M and its concurrent encoding Γ. Then for all states p and q and strings x, $p \xrightarrow{x} q$ if and only if $\Gamma; \cdot \vdash \forall y.\, p(x \cdot y) \multimap (q(y) \otimes \top)$. In particular, if M has initial state s and final state f, then M accepts x if and only if $\Gamma; \cdot \vdash s(x) \multimap (f(\epsilon) \otimes \top)$.

While this expresses correctness, it does not explicitly address the concurrency aspects. For example, even our prior encoding would satisfy this requirement even though it does not encode any concurrency. We omit the hypothesis labels in this encoding.

Cases: $r = a$ or $r = r_1 \cdot r_2$ or $r = \mathbf{1}$. As before.

Case: $r = r_1 + r_2$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translations of r_1 and r_2 respectively. We construct

$$\begin{array}{rcl}\Gamma(s,f) & = & \forall x.\, s(x) \multimap (s_1(x) \otimes s_2(x)),\\ & & \Gamma_1(s_1,f_1), \Gamma_2(s_2,f_2),\\ & & \forall y.\, f_1(y) \multimap f(y),\\ & & \forall y.\, f_2(y) \multimap f(y)\end{array}$$

Now there is no alternative formulation of the first rule.

Case: $r = \mathbf{0}$ where $\mathcal{L}(\mathbf{0}) = \{\,\}$.

$$\Gamma(s,f) \quad = \quad \forall x.\, s(x) \multimap \mathbf{1}$$

with no rule for f.

Case: $r = r_1^*$ where $\mathcal{L}(r_1^*) = \{x_1 \cdots x_n \mid x_i \in \mathcal{L}(r_1) \text{ for } 1 \leq i \leq n \text{ and } n \geq 0\}$. Let $\Gamma_1(s_1, f_1)$ be the translation of r_1. Then we construct

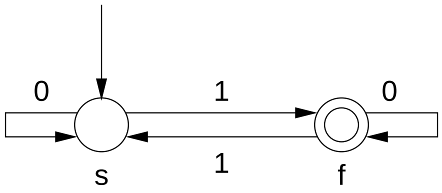
$$\begin{array}{rcl}\Gamma(s,f) & = & \forall x.\, s(x) \multimap (f(x) \otimes s_1(x)),\\ & & \Gamma_1(s_1,f_1),\\ & & \forall y.\, f_1(y) \multimap s(y)\end{array}$$

Cases: $r = r_1 \cap r_2$ and $r_1 = \mathbf{T}$. As before.

This form of concurrent encoding requires some consideration of scheduling the processes represented by linear hypotheses. For example, if we have a regular expression $\epsilon^* \cdot a$ we have to be careful not to schedule the process corresponding to ϵ^* indefinitely without scheduling the process for a, or we may never accept the string a.

These issues foreshadow similar considerations for more complex concurrent systems in linear logic later on. Note that computation in this setting corresponds to a kind of forward-chaining proof search. Other models of computation are also possible and often appropriate. In particular, we may describe computation via backward-chaining search, or by proof reduction. We close this section by showing a backward-chaining implementation of finite automata and regular expressions.

Recall the simple automaton to accept binary strings with odd parity.



In the coding we now simply *reverse* all the arrows.

$$\begin{array}{lcl} s^0 & : & \forall x.\, s(0 \cdot x) \mathrel{\circ\!-} s(x) \\ s^1 & : & \forall x.\, s(1 \cdot x) \mathrel{\circ\!-} f(x) \\ f^0 & : & \forall x.\, f(0 \cdot x) \mathrel{\circ\!-} f(x) \\ f^1 & : & \forall x.\, f(1 \cdot x) \mathrel{\circ\!-} s(x) \end{array}$$

Then, the automaton accepts x if and only if we can prove

$$f(\epsilon) \multimap s(x),$$

again reversing the arrow from the previous statement where we prove $s(x) \multimap f(\epsilon)$ instead.

Adequacy for Backward-Chaining Encoding of Automata.

> Given a non-deterministic finite automaton M and its encoding Γ. Then for all states p and q and strings x, $p \xrightarrow{x} q$ if and only if $\Gamma; \cdot \vdash \forall y.\, q(y) \multimap p(x \cdot y)$. In particular, if M has initial state s and final state f, then M accepts x if and only if $\Gamma; \cdot \vdash f(\epsilon) \multimap s(x)$.

For completness, we now give the backward-chaining encoding of regular expressions.

Case: $r = c$ for a character c. Then

$$\Gamma(s, f) = \forall y.\, s(c \cdot y) \mathrel{\circ\!-} f(y)$$

Case: $r = r_1 \cdot r_2$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translations of r_1 and r_2 respectively. We construct

$$\begin{array}{lcl} \Gamma(s, f) & = & \forall x.\, s(x) \mathrel{\circ\!-} s_1(x), \\ & & \Gamma_1(s_1, f_1), \\ & & \forall z.\, f_1(z) \mathrel{\circ\!-} s_2(z), \\ & & \Gamma_2(s_2, f_2), \\ & & \forall y.\, f_2(y) \mathrel{\circ\!-} f(y) \end{array}$$

Case: $r = \mathbf{1}$ where $\mathcal{L}(\mathbf{1}) = \{\epsilon\}$. Then

$$\Gamma(s, f) = \forall y.\, s(y) \mathrel{\circ\!-} f(y)$$

Case: $r = r_1 + r_2$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translations of r_1 and r_2 respectively. We construct

$$\begin{array}{lcl} \Gamma(s, f) & = & \forall x.\, s(x) \mathrel{\circ\!-} (s_1(x) \oplus s_2(x)), \\ & & \Gamma_1(s_1, f_1), \Gamma_2(s_2, f_2), \\ & & \forall y.\, f_1(y) \mathrel{\circ\!-} f(y), \\ & & \forall y.\, f_2(y) \mathrel{\circ\!-} f(y) \end{array}$$

Interestingly, the internal choice $s_1(x)\&s_2(x)$ is turned into an external choice $s_1(x) \oplus s_2(x)$ when we turn a forward chaining to a backward chaining implementation. Again, there are some alternatives. For example, the last two propositions can be combined into

$$\forall y.\, (f_1(y)\&f_2(y)) \circ\!- f(y)$$

Case: $r = \mathbf{0}$ where $\mathcal{L}(\mathbf{0}) = \{\,\}$.

$$\Gamma(s, f) \;=\; \forall x.\, s(x) \circ\!- \mathbf{0}$$

with no rule for f. Again, alternatives are possible.

Case: $r = r_1^*$. Let $\Gamma_1(s_1, f_1)$ be the translation of r_1. Then we construct

$$\begin{array}{rcl} \Gamma(s, f) & = & \forall x.\, s(x) \circ\!-(f(x) \oplus s_1(x)), \\ & & \Gamma_1(s_1, f_1), \\ & & \forall y.\, f_1(y) \circ\!- s(y) \end{array}$$

Case: $r = r_1 \cap r_2$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translation of r_1 and r_2. This case appears to be quite tricky, because of the lack of any natural concurrency in the backward-chaining model.[1]

Case: $r = \mathbf{T}$. Then

$$\Gamma(s, f) \;=\; \forall x.\, \forall y.\, s(x \cdot y) \circ\!- f(y)$$

We now return to our first encoding of regular expressions. How can we prove the adequacy of the representation? Recall the statement of the adequacy theorem.

Adequacy for Encoding of Regular Expressions.

Given a regular expression r and its encoding Γ with distinguished predicates s (*start*) and f (*final*). Then $x \in \mathcal{L}(r)$ if and only if $\Gamma; \cdot \vdash \forall y.\, s(x \cdot y) \multimap f(y)$.

We first prove that if $x \in \mathcal{L}(r)$, then $\Gamma(s, f); \cdot \vdash \forall y.\, s(x \cdot y) \multimap f(y)$. In all cases we reduce this to proving

$$\Gamma(s, f); s(x \cdot y) \vdash f(y)$$

for a new parameter y. The adequacy follows from this in two steps by $\multimap$I and $\forall$I. The proof is now by induction on the structure of r. We restate the translations, this time giving explicit labels to assumptions so we can refer to them in the proof.

[1] Suggestions welcome!

Case: $r = a$ for a character a where $\mathcal{L}(a) = \{a\}$. Then

$$\Gamma(s, f) = v_s : \forall y.\ s(a \cdot y) \multimap f(y)$$

So we have to show

$$v_s : \forall y.\ s(a \cdot y) \multimap f(y); s(a \cdot y) \vdash f(y)$$

which follows in three steps.

$v_s : \forall y.\ s(a \cdot y) \multimap f(y); s(a \cdot y) \vdash s(a \cdot y)$	Linear hypothesis
$v_s : \forall y.\ s(a \cdot y) \multimap f(y); \cdot \vdash \forall y.\ s(a \cdot y) \multimap f(y)$	Unrestricted hypothesis
$v_s : \forall y.\ s(a \cdot y) \multimap f(y); \cdot \vdash s(a \cdot y) \multimap f(y)$	By rule $\forall$E
$v_s : \forall y.\ s(a \cdot y) \multimap f(y); s(a \cdot y) \vdash f(y)$	By rule $\multimap$E

Case: $r = r_1 \cdot r_2$ where $\mathcal{L}(r_1 \cdot r_2) = \{x_1 \cdot x_2 \mid x_1 \in \mathcal{L}(r_1) \text{ and } x_2 \in \mathcal{L}(r_2)\}$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translations of r_1 and r_2 respectively. We construct

$$\begin{array}{rcl} \Gamma(s, f) & = & v_s : \forall x.\ s(x) \multimap s_1(x), \\ & & \Gamma_1(s_1, f_1), \\ & & v_{f_1} : \forall z.\ f_1(z) \multimap s_2(z), \\ & & \Gamma_2(s_2, f_2), \\ & & v_{f_2} : \forall y.\ f_2(y) \multimap f(y) \end{array}$$

We have to show

$$\Gamma(s, f); s(x_1 \cdot x_2 \cdot y) \vdash f(y)$$

for a new parameter y.

$\Gamma(s, f); s(x_1 \cdot x_2 \cdot y) \vdash s(x_1 \cdot x_2 \cdot y)$	Linear hypothesis
$\Gamma(s, f); s(x_1 \cdot x_2 \cdot y) \vdash s_1(x_1 \cdot x_2 \cdot y)$	From v_s by $\forall$E and $\multimap$E
$\Gamma(s, f); s(x_1 \cdot x_2 \cdot y) \vdash f_1(x_2 \cdot y)$	By i.h. on r_1, weakening and subst.
$\Gamma(s, f); s(x_1 \cdot x_2 \cdot y) \vdash s_2(x_2 \cdot y)$	From v_{f_1} by $\forall$E and $\multimap$E
$\Gamma(s, f); s(x_1 \cdot x_2 \cdot y) \vdash f_2(y)$	By i.h. on r_2, weakening and subst.
$\Gamma(s, f); s(x_1 \cdot x_2 \cdot y) \vdash f(y)$	from v_{f_2} by $\forall$E and $\multimap$E

Case: $r = \mathbf{1}$ where $\mathcal{L}(\mathbf{1}) = \{\epsilon\}$. Then

$$\Gamma(s, f) = v_s : \forall y.\ s(y) \multimap f(y)$$

This case is trivial, since $\epsilon \cdot y = y$.

$\Gamma(s, f); s(\epsilon \cdot y) \vdash s(\epsilon \cdot y)$	Linear hypothesis
$\Gamma(s, f); s(\epsilon \cdot y) \vdash f(y)$	From v_s by $\forall$E, $\multimap$E since $\epsilon \cdot y = y$

Case: $r = r_1 + r_2$ where $\mathcal{L}(r_1 + r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translations of r_1 and r_2 respectively. We construct

$$\begin{array}{rcl} \Gamma(s, f) & = & v_s : \forall x.\ s(x) \multimap (s_1(x) \& s_2(x)), \\ & & \Gamma_1(s_1, f_1), \Gamma_2(s_2, f_2), \\ & & v_{f_1} : \forall y.\ f_1(y) \multimap f(y), \\ & & v_{f_2} : \forall y.\ f_2(y) \multimap f(y) \end{array}$$

Let $x \in \mathcal{L}(r)$. Then there are two subcases. We show the case where $x \in \mathcal{L}(r_1)$; the other subcase is symmetric.

$$\begin{array}{ll}
\Gamma(s,f); s(x \cdot y) \vdash s(x \cdot y) & \text{Linear hypothesis} \\
\Gamma(s,f); s(x \cdot y) \vdash s_1(x \cdot y) \& s_2(x \cdot y) & \text{From } v_s \text{ by } \forall\text{E}, \multimap\text{E} \\
\Gamma(s,f); s(x \cdot y) \vdash s_1(x \cdot y) & \text{By rule } \&\text{E}_\text{L} \\
\Gamma(s,f); s(x \cdot y) \vdash f_1(y) & \text{By i.h. on } r_1\text{, weakening and subst.} \\
\Gamma(s,f); s(x \cdot y) \vdash f(y) & \text{From } v_{f_1} \text{ by } \forall\text{E}, \multimap\text{E}
\end{array}$$

Case: $r = \mathbf{0}$ where $\mathcal{L}(\mathbf{0}) = \{\,\}$.

$$\Gamma(s,f) \quad = \quad v_s : \forall x.\, s(x) \multimap \top$$

with no rule for f. Then the conclusion follows trivially since there is no $x \in \mathcal{L}(\mathbf{0})$.

Case: $r = r_1^*$ where $\mathcal{L}(r_1^*) = \{x_1 \cdots x_n \mid x_i \in \mathcal{L}(r_1) \text{ for } 1 \leq i \leq n \text{ and } n \geq 0\}$.

$$\begin{array}{rcl}
\Gamma(s,f) & = & v_s : \forall x.\, s(x) \multimap (f(x) \& s_1(x)), \\
 & & \Gamma_1(s_1, f_1), \\
 & & v_{f_1} : \forall y.\, f_1(y) \multimap s(y)
\end{array}$$

Assume $x = x_1 \cdots x_n \in \mathcal{L}(r_1^*)$ where each $x_i \in r_1$ for $1 \leq i \leq n$. We conduct an auxiliary induction on n.

Subcase: $n = 0$. Then, by definition, $x_1 \cdots x_n = \epsilon$.

$$\begin{array}{ll}
\Gamma(s,f); s(\epsilon \cdot y) \vdash s(\epsilon \cdot y) & \text{Linear hypothesis} \\
\Gamma(s,f); s(\epsilon \cdot y) \vdash f(y) \& s_1(x) & \text{From } v_s \text{ by } \forall\text{E}, \multimap\text{E} \\
\Gamma(s,f); s(\epsilon \cdot y) \vdash f(y) & \text{By rule } \&\text{E}_\text{L}
\end{array}$$

Subcase: $n > 0$. Then

$$\begin{array}{ll}
\Gamma(s,f); s(x_1 \cdots x_n \cdot y) \vdash s(x_1 \cdots x_n \cdot y) & \text{Linear hypothesis} \\
\Gamma(s,f); s(x_1 \cdots x_n \cdot y) \vdash f(x_1 \cdots x_n \cdot y) \& s_1(x_1 \cdots x_n \cdot y) & \text{From } v_s \\
\Gamma(s,f), s(x_1 \cdots x_n \cdot y) \vdash s_1(x_1 \cdots x_n \cdot y) & \text{By rule } \&\text{E}_\text{R} \\
\Gamma(s,f), s(x_1 \cdots x_n \cdot y) \vdash f_1(x_2 \cdots x_n \cdot y) & \text{By i.h. since } x_1 \in \mathcal{L}(r_1) \\
\Gamma(s,f), s(x_1 \cdots x_n \cdot y) \vdash s(x_2 \cdots x_n \cdot y) & \text{From } v_{f_1} \\
\Gamma(s,f), s(x_1 \cdots x_n \cdot y) \vdash f(y) & \text{From i.h. on } n-1
\end{array}$$

We now also show the correctness for the encoding of intersection and all strings.

Case: $r = r_1 \cap r_2$ where $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$. Let $\Gamma_1(s_1, f_1)$ and $\Gamma_2(s_2, f_2)$ be the translation of r_1 and r_2.

$$\begin{array}{rcl}
\Gamma(s,f) & = & v_s : \forall x.\, s(x) \multimap (s_1(x) \otimes s_2(x)), \\
 & & \Gamma_1(s_1, f_1), \Gamma_2(s_2, f_2), \\
 & & v_{f_{12}} : \forall y.\, (f_1(y) \otimes f_2(y)) \multimap f(y)
\end{array}$$

Assume $x \in \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$. Then

$\Gamma(s,f); s(x \cdot y) \vdash s(x \cdot y)$	Linear hypothesis
$\Gamma(s,f); s(x \cdot y) \vdash s_1(x \cdot y) \otimes s_2(x \cdot y)$	From v_s
$\Gamma(s,f); s_1(x \cdot y) \vdash f_1(y)$	From i.h. since $x \in \mathcal{L}(r_1)$
$\Gamma(s,f); s_2(x \cdot y) \vdash f_2(y)$	From i.h. since $x \in \mathcal{L}(r_2)$
$\Gamma(s,f); s_1(x \cdot y), s_2(x \cdot y) \vdash f_1(y) \otimes f_2(y)$	By $\otimes$I
$\Gamma(s,f); s(x \cdot y) \vdash f_1(y) \otimes f_2(y)$	By $\otimes$E
$\Gamma(s,f); s(x \cdot y) \vdash f(y)$	From $v_{f_{12}}$

Case: $r = \mathbf{T}$ where $\mathcal{L}(\mathbf{T}) = \Sigma^*$, the set of all strings over the alphabet Σ.

$$\Gamma(s,f) \quad = \quad v_s : \forall x.\, \forall y.\, s(x \cdot y) \multimap f(y)$$

Let x by an arbitrary string. Then

$\Gamma(s,f); s(x \cdot y) \vdash s(x \cdot y)$	Linear hypothesis
$\Gamma(s,f); s(x \cdot y) \vdash f(y)$	From v_s

The other direction is much harder to prove. Assume we have a regular expression r, its encoding $\Gamma(s,f)$, and a deduction of $\Gamma; \cdot \forall y.\, s(x \cdot y) \multimap f(y)$. We need to show that $x \in \mathcal{L}(r)$. In order to do this, we need to analyse the structure of the given deduction. In some sense, we are showing that the deduction we gave in the other direction above are "inevitable". To illustrate the difficulty, consider, for example, the $\multimap$ rule.

$$\frac{\Gamma; \Delta_1 \vdash A \multimap B \qquad \Gamma; \Delta_2 \vdash A}{\Gamma; (\Delta_1, \Delta_2) \vdash B} \multimap E$$

If we are trying to prove B from Γ and Δ, two problems arise. First, we have to decide how to split Δ into Δ_1 and Δ_2. More importantly, however, how do we choose A?

If we look back at the development of our logic, we introduced this rule to answer the question "*How do we use the knowledge that $A \multimap B$ true*". Above, however, we think about how to prove B. It is this mismatch which makes rules like $\multimap$E intractable.

In the next section we will introduce a restriction on the application of the inference rules whereby introduction rules are only applied bottom-up while elimination rules are applied only top-down. With this restriction it will be possible to show to prove the more difficult direction of adequacy for the encoding of regular expressions.

2.6 Normal Deductions

The judgmental approach to understanding propositions and truth defines the meaning of the connectives by giving the introduction and elimination rules. We claim, for example, that $A \& B$ *true* if A *true* and B *true*. We see this as a defining property for alternative conjunction (once the hypotheses have been added to the rule). But does our final deductive system validate this interpretation of alternative conjunction? For example, it could be the case $C \multimap (A \& B)$ *true* and C *true* but there is no direct way of deriving A *true* and B *true* without the detour through C.

Local soundness and completeness are important tools to verify the correctness of our system. They verify that if we introduce and then immediately eliminate a connectives, this detour can be avoided. This is a local property of a derivation. However, it is possible that we introduce a connective and eliminate it at some later point in a proof, but not immediately. For example,

$$\dfrac{\dfrac{\dfrac{}{A \otimes B \Vdash A \otimes B}\,\mathsf{hyp} \qquad \dfrac{\begin{matrix}\mathcal{D}\\ A, B \Vdash C\end{matrix} \qquad \begin{matrix}\mathcal{E}\\ A, B \Vdash D\end{matrix}}{A, B \Vdash C \& D}\,\&\mathrm{I}}{A \otimes B \Vdash C \& D}\,\otimes\mathrm{E}}{A \otimes B \Vdash C}\,\&\mathrm{E_L}$$

This deduction contains a detour, since we first introduce $C\&D$ and then later eliminate it. In a derivation of this form, we cannot carry out a local reduction because $C\&D$ is introduced above and eliminated below the application $\otimes$E. In this case it is easy to see how to correct the problem: we move the application of &E to come above the application of $\otimes$E, and then carry out a local reduction.

$$\dfrac{\dfrac{}{A \otimes B \Vdash A \otimes B}\,\mathsf{hyp} \qquad \dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ A, B \Vdash C\end{matrix} \qquad \begin{matrix}\mathcal{E}\\ A, B \Vdash D\end{matrix}}{A, B \Vdash C \& D}\,\&\mathrm{I}}{A, B \Vdash C}\,\&\mathrm{E_L}}{A \otimes B \Vdash C}\,\otimes\mathrm{E}$$

This then reduces to

$$\dfrac{\dfrac{}{A \otimes B \Vdash A \otimes B}\,\mathsf{hyp} \qquad \begin{matrix}\mathcal{D}\\ A, B \Vdash C\end{matrix}}{A \otimes B \Vdash C}\,\otimes\mathrm{E}$$

What we eventually want to show is global soundness and completeness.

Global soundness states that every evident judgment $\Gamma; \Delta \vdash A$ *true* has a derivation in which we only apply introduction rules to conclusions that we are trying to prove and elimination rules to hypotheses or consequences we have derived from them. We all such derivations *normal.* Normal derivations have the important *subformula property*: every judgment occurring in a normal derivation of $\Gamma; \Delta \vdash A$ *true* refers only to subformulas of Γ, Δ, and A. This means our definition of truth is internally consistent and well-founded. It also means that our connectives are orthogonal to each other: we can understand each connective in isolation, falling back only on judgmental notions in their definition.

Global completeness means that every evident judgment $\Gamma; \Delta \vdash A\ \mathit{true}$ has a derivation in which every conclusion is eventually inferred by an introduction rule. For this to be the case, the elimination rules need to be strong enough so we can decompose our hypothesis and reassemble the conclusion from the atomic constituents, where a proposition is atomic if it doesn't have a top-level logical connective. We call such a derivation a *long normal* derivation, because it corresponds to the notion of *long normal form* in λ-calculi.

In order to prove these properties for our logic, we need to define more formally what *normal* and *long normal* deductions are. We postpone the discussion of long normal derivation and just concentrate on normal derivations in this section. We express this by two mutually recursive judgments that reflect the nature of hypothetical reasoning with introduction and elimination rules.

$A\uparrow$	A has a normal derivation
$A\downarrow$	A has an atomic derivation

The idea that an atomic derivation is either a direct use of a hypothesis (either linear or unrestricted), or the result of applying an elimination rule to an atomic derivation. We therefore consider only hypotheses of the form $A\downarrow$ (for linear hypotheses) and $A\downdownarrows$ (for unrestricted hypotheses). Thus, we consider hypothetical judgments linear in $A\downarrow$ and unrestricted in $A\downdownarrows$.

$$(v_1{:}B_1\downdownarrows, \ldots v_m{:}B_m\downdownarrows); (u_1{:}A_1\downarrow, \ldots u_n{:}A_n\downarrow) \vdash A\uparrow$$
$$(v_1{:}B_1\downdownarrows, \ldots v_m{:}B_m\downdownarrows); (u_1{:}A_1\downarrow, \ldots u_n{:}A_n\downarrow) \vdash A\downarrow$$

We abbreviate the unrestricted hypotheses with Γ and linear hypotheses with Δ, but we should keep in mind that they stand for assumptions of the form $A\downdownarrows$ and $A\downarrow$, respectively.

These formalize an intuitive strategy in constructing natural deductions is to apply introduction rules backwards to break the conclusion into subgoals and to apply elimination rules to hypotheses until the two meet. These judgments are defined by the following inference rules.

Hypotheses.

$$\frac{}{\Gamma; u{:}A\downarrow \vdash A\downarrow}\,u \qquad \frac{}{(\Gamma, v{:}A\downdownarrows); \cdot \vdash A\downarrow}\,v$$

Multiplicative Connectives.

$$\frac{\Gamma; \Delta_1 \vdash A\uparrow \qquad \Gamma; \Delta_2 \vdash B\uparrow}{\Gamma; (\Delta_1, \Delta_2) \vdash A \otimes B\uparrow}\,\otimes\mathrm{I}$$

$$\frac{\Gamma; \Delta \vdash A \otimes B\downarrow \quad \Gamma; (\Delta', u{:}A\downarrow, w{:}B\downarrow) \vdash C\uparrow}{\Gamma; (\Delta, \Delta') \vdash C\uparrow}\,\otimes\mathrm{E}$$

$$\frac{\Gamma; (\Delta, u{:}A\downarrow) \vdash B\uparrow}{\Gamma; \Delta \vdash A \multimap B\uparrow} \multimap \mathrm{I} \qquad \frac{\Gamma; \Delta \vdash A \multimap B\downarrow \qquad \Gamma; \Delta' \vdash A\uparrow}{\Gamma; \Delta, \Delta' \vdash B\downarrow} \multimap \mathrm{E}$$

$$\frac{}{\Gamma; \cdot \vdash \mathbf{1}\uparrow} \mathbf{1}\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash \mathbf{1}\downarrow \qquad \Gamma; \Delta' \vdash C\uparrow}{\Gamma; (\Delta, \Delta') \vdash C\uparrow} \mathbf{1}\mathrm{E}$$

Additive Connectives.

$$\frac{\Gamma; \Delta \vdash A\uparrow \qquad \Gamma; \Delta \vdash B\uparrow}{\Gamma; \Delta \vdash A \& B\uparrow} \&\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash A \& B\downarrow}{\Gamma; \Delta \vdash A\downarrow} \&\mathrm{E}_{\mathrm{L}} \qquad \frac{\Gamma; \Delta \vdash A \& B\downarrow}{\Gamma; \Delta \vdash B\downarrow} \&\mathrm{E}_{\mathrm{R}}$$

$$\frac{}{\Gamma; \Delta \vdash \top\uparrow} \top\mathrm{I} \qquad \textit{No } \top \textit{ elimination rule}$$

$$\frac{\Gamma; \Delta \vdash A\uparrow}{\Gamma; \Delta \vdash A \oplus B\uparrow} \oplus\mathrm{I}_{\mathrm{L}} \qquad \frac{\Gamma; \Delta \vdash B\uparrow}{\Gamma; \Delta \vdash A \oplus B\uparrow} \oplus\mathrm{I}_{\mathrm{R}}$$

$$\frac{\Gamma; \Delta \vdash A \oplus B\downarrow \quad \Gamma; (\Delta', u{:}A\downarrow) \vdash C\uparrow \quad \Gamma; (\Delta', w{:}B\downarrow) \vdash C\uparrow}{\Gamma; (\Delta, \Delta') \vdash C\uparrow} \oplus E$$

$$\textit{No } \mathbf{0} \textit{ introduction rule} \qquad \frac{\Gamma; \Delta \vdash 0\downarrow}{\Gamma; (\Delta, \Delta') \vdash C\uparrow} \mathbf{0}\mathrm{E}$$

Quantifiers.

$$\frac{\Gamma; \Delta \vdash [a/x]A\uparrow}{\Gamma; \Delta \vdash \forall x.\ A\uparrow} \forall\mathrm{I}^a \qquad \frac{\Gamma; \Delta \vdash \forall x.\ A\downarrow}{\Gamma; \Delta \vdash [t/x]A\downarrow} \forall\mathrm{E}$$

$$\frac{\Gamma; \Delta \vdash [t/x]A\uparrow}{\Gamma; \Delta \vdash \exists x.\ A\uparrow} \exists\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash \exists x.\ A\downarrow \qquad \Gamma; (\Delta', u{:}[a/x]A\downarrow) \vdash C\uparrow}{\Gamma; (\Delta', \Delta) \vdash C\uparrow} \exists\mathrm{E}^a$$

Exponentials.

$$\frac{(\Gamma, v{:}A\downarrow\!\downarrow); \Delta \vdash B\uparrow}{\Gamma; \Delta \vdash A \supset B\uparrow} \supset\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash A \supset B\downarrow \qquad \Gamma; \cdot \vdash A\uparrow}{\Gamma; \Delta \vdash B\downarrow} \supset\mathrm{E}$$

$$\frac{\Gamma; \cdot \vdash A \uparrow}{\Gamma; \cdot \vdash {!A} \uparrow} !I \qquad \frac{\Gamma; \Delta \vdash {!A} \downarrow \qquad (\Gamma, v{:}A \downdownarrows); \Delta' \vdash C \uparrow}{\Gamma; (\Delta', \Delta) \vdash C \uparrow} !\mathrm{E}$$

Coercion.

$$\frac{\Gamma; \Delta \vdash A \downarrow}{\Gamma; \Delta \vdash A \uparrow} \downarrow\uparrow$$

The coercion $\downarrow\uparrow$ states that all atomic derivations should be considered normal. From the point of view of proof search this means that we can complete the derivation when forward and backward reasoning arrive at the same proposition. We obtain long normal derivation if we restrict the coercion rule to atomic propositions. It easy to see that these judgments just restrict the set of derivations.

Property 2.1 (Soundness of Normal Derivations)

1. *If* $\Gamma; \Delta \vdash A \uparrow$ *then* $\Gamma; \Delta \vdash A$.
2. *If* $\Gamma; \Delta \vdash A \downarrow$ *then* $\Gamma; \Delta \vdash A$.

Proof: *By simultaneous induction on the given derivations. The computational contents of this proof are the obvious structural translation from* $\mathcal{N} :: (\Gamma; \Delta \vdash A \uparrow)$ *to* $\mathcal{N}^- :: (\Gamma; \Delta \vdash A)$ *and from* $\mathcal{A} :: (\Gamma; \Delta \vdash A \downarrow)$ *to* $\mathcal{A}^- :: (\Gamma; \Delta \vdash A)$. *Note that the coercion* $\downarrow\uparrow$ *disappears, since the translation of the premise and conclusion are identical.* □

The corresponding completeness theorem, namely that $\Gamma; \Delta \vdash A$ implies $\Gamma; \Delta \vdash A \uparrow$, also holds, but is quite difficult to prove. This is the subject of the Normalization Theorem 3.10. Together with the two judgments about atomic and normal derivations, we have refined substitution principles. Since hypotheses are atomic, they permit only the substitution of atomic derivations for hypotheses.[2]

Lemma 2.2 (Substitution Principles for Atomic Derivations)

1. *If* $\Gamma; \Delta \vdash A \downarrow$ *and* $\Gamma; (\Delta', u{:}A \downarrow) \vdash C \uparrow$ *then* $\Gamma; (\Delta, \Delta') \vdash C \uparrow$.
2. *If* $\Gamma; \Delta \vdash A \downarrow$ *and* $\Gamma; (\Delta', u{:}A \downarrow) \vdash C \downarrow$ *then then* $\Gamma; (\Delta, \Delta') \vdash C \downarrow$.
3. *If* $\Gamma; \cdot \vdash A \downarrow$ *and* $(\Gamma, v{:}A \downdownarrows); \Delta \vdash C \uparrow$ *and then* $\Gamma; \Delta \vdash C \uparrow$.
4. *If* $\Gamma; \cdot \vdash A \downarrow$ *and* $(\Gamma, v{:}A \downdownarrows); \Delta \vdash C \downarrow$ *and then* $\Gamma; \Delta \vdash C \downarrow$.

Proof: By straightforward inductions over the structure of the derivation we substitute into, appealing to weakening and exchange properties. □

[2] We have not formally stated the substitution principles for natural deduction as theorems of the complete system. They can be proven easily by induction on the structure of the derivation we substitute into.

2.7 Exercises

Exercise 2.1 Give a counterexample that shows that the restriction to empty linear hypotheses in the substitution principle for validity is necessary.

Exercise 2.2 Give a counterexample which shows that the elimination $\supset$E would be locally unsound if its second premise were allowed to depend on linear hypotheses.

Exercise 2.3 If we *define* unrestricted implication $A \supset B$ in linear logic as an abbreviation for $(!A) \multimap B$, then the given introduction and elimination rules become *derived rules of inference*. Prove this by giving a derivation for the conclusion of the $\supset$E rule from its premises under the interpretation, and similarly for the $\supset$I rule.

For the other direction, show how $!A$ could be defined from unrestricted implication or speculate why this might not be possible.

Exercise 2.4 Speculate bout the "missing connectives" of multiplicative disjunction, multiplicative falsehood, and additive implication. What would the introduction and elimination rules look like? What is the difficulty? any ideas for how these difficulties might be overcome?

Exercise 2.5 In the blocks world example, sketch the derivation for the same goal A_0 and initial situation Δ_0 in which block b is put on block c, rather than the table.

Exercise 2.6 Model the *Towers of Hanoi* in linear logic in analogy with our modelling of the blocks world.

1. Define the necessary atomic propositions and their meaning.
2. Describe the legal moves in *Towers of Hanoi* as unrestricted hypotheses Γ_0 independently from the number of towers or disks.
3. Represent the initial situation of three towers, where two are empty and one contains two disks in a legal configuration.
4. Represent the goal of legally stacking the two disks on some arbitrary other tower.
5. Sketch the proof for the obvious 3-move solution.

Exercise 2.7 Consider if $\otimes$ and $\&$ can be distributed over $\oplus$ or *vice versa*. There are four different possible equivalences based on eight possible entailments. Give natural deductions for the entailments which hold.

Exercise 2.8 In this exercise we explore distributive and related *interaction laws* for linear implication. In intuitionistic logic, for example, we have the following $(A \wedge B) \supset C \dashv\vdash A \supset (B \supset C)$ and $A \supset (B \wedge C) \dashv\vdash (A \supset B) \wedge (A \supset C)$, where $\dashv\vdash$ is mutual entailment as in Exercise **??**.

In linear logic, we now write $A \dashv\vdash A'$ for linear mutual entailment, that is, A' follows from linear hypothesis A and *vice versa*. Write out appropriate interaction laws or indicate none exists, for each of the following propositions.

1. $A \multimap (B \otimes C)$
2. $(A \otimes B) \multimap C)$
3. $A \multimap \mathbf{1}$
4. $\mathbf{1} \multimap A$
5. $A \multimap (B \& C)$
6. $(A \& B) \multimap C$
7. $A \multimap \top$
8. $\top \multimap A$
9. $A \multimap (B \oplus C)$
10. $(A \oplus B) \multimap C$
11. $A \multimap \mathbf{0}$
12. $\mathbf{0} \multimap A$
13. $A \multimap (B \multimap C)$
14. $(A \multimap B) \multimap C$

Note that an interaction law exists only if there is a mutual linear entailment—we are not interested if one direction holds, but not the other.

Give the derivations in both directions for one of the interaction laws of a binary connective $\otimes$, $\&$, $\oplus$, or $\multimap$, and for one of the interaction laws of a logical constant $\mathbf{1}$, $\top$, or $\mathbf{0}$.

Exercise 2.9 Consider three forms of equivalence of propositions in linear logic.

- $A \multimapboth B$ which should be true if A linearly implies B and vice versa.
- $A \simeq B$ which should be true if, independently of any linear hypotheses, A linearly implies B and vice versa.
- $A \equiv B$ which should be true if A implies B and B implies A, where both implications are unrestricted.

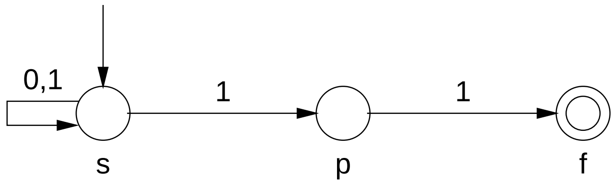
1. For each of the these connectives, give introduction and elimination rules and show local soundness and completeness of your rules. If it is not possible, argue why. Be careful that your rules do *not* refer to other connectives, but rely entirely on judgmental concepts.

2. Discuss if the specification above is unambiguous or if interpretations essentially different from yours may be possible.

3. Using your rules, prove each linear entailment $A \ \mathrm{op}_1\ B\ \mathit{true} \Vdash A\ \mathrm{op}_2\ B\ \mathit{true}$ that holds where op_i are equivalence operators.

4. [Extra Credit] Give counterexamples for the entailments that do not hold.

Exercise 2.10 Consider the non-deterministic finite automaton



where s is the start state and f is the final (accepting) state. Represent this automaton in linear logic according to two strategies: one where non-deterministic choice is modeled as internal choice ($\&$), and one where non-deterministic choice is modeled as concurrency ($\otimes$). Show the proof corresponding to the accepting computation for the string $0 \cdot 1 \cdot 1 \cdot 1$ in both encodings.

Chapter 3

Sequent Calculus

In the previous chapter we developed linear logic in the form of natural deduction, which is appropriate for many applications of linear logic. It is also highly economical, in that we only needed one basic judgment (A *true*) and two judgment forms (linear and unrestricted hypothetical judgments) to explain the meaning of all connectives we have encountered so far. However, it is not immediately well-suited for proof search, because it involves mixing forward and backward reasoning even if we restrict ourselves to searching for normal deductions.

In this chapter we develop a sequent calculus as a calculus of proof search for normal natural deductions. We then extend it with a rule of cut that allows us to model arbitrary natural deductions. The central theorem of this chapter is *cut elimination* which shows that the cut rule is admissible. We obtain the normalization theorem for natural deduction as a direct consequence of this theorem. It was this latter application which led to the original discovery of the sequent calculus by Gentzen [Gen35]. There are many useful immediate corollaries of the cut elimination theorem, such as consistency of the logic, or the disjunction property.

3.1 Cut-Free Sequent Calculus

In this section we transcribe the process of searching for *normal* natural deductions into an inference system. In the context of sequent calculus, proof search is seen entirely as the bottom-up construction of a derivation. This means that elimination rules must be turned "upside-down" so they can also be applied bottom-up rather than top-down.

In terms of judgments we develop the sequent calculus via a splitting of the judgment "*A is true*" into two judgments: "*A is a resource*" (A *res*) and "*A is a goal*" (A *goal*). Ignoring unrestricted hypothesis for the moment, the main judgment

$$w_1{:}A_1\ \mathit{res}, \ldots, w_n{:}A_n\ \mathit{res} \Longrightarrow C\ \mathit{goal}$$

expresses

> Under the linear hypothesis that we have resources $A_1, \ldots, A_n$ we can achieve goal C.

In order to model validity, we add inexhaustible resources or *resource factories*, written A *fact*. We obtain

$$(v_1{:}B_1\,\mathit{fact}, \ldots, v_m{:}B_m\,\mathit{fact}); (w_1{:}A_1\ \mathit{res}, \ldots, w_n{:}A_n\ \mathit{res}) \Longrightarrow C\ \mathit{goal},$$

which expresses

> Under the unrestricted hypotheses that we have resource factories $B_1, \ldots, B_m$ and linear hyptheses that we have resources $A_1, \ldots, A_n$, we can achieve goal C.

As before, the order of the hypothesis (linear or unrestricted) is irrelevant, and we assume that all hypothesis labels v_j and w_i are distinct.

Resources and goals are related in that with the resource A we can achieve goal A. Recall that the linear hypothetical judgment requires us to use all linear hypotheses exactly once. We therefore have the following rule.

$$\frac{}{\Gamma; w{:}A\ \mathit{res} \Longrightarrow A\ \mathit{goal}}\ \mathsf{init}_w$$

We call such as sequent *initial* and write init. Note that, for the moment, we do not have the opposite: if we can achieve goal A we cannot assume A as a resource. The corresponding rule will be called cut and is shown later to be admissible, that is, every instance of this rule can be eliminated from a proof. It is the desire to rule out cut that necessitated splitting truth into two judgments.

Note that the initial rule does *not* follow directly from the nature of linear hypothetical judgments, since A *res* and A *goal* are different judgments. Instead, it explicitly states a connection between resources and goals. A rule that concludes $\Gamma; A\ \mathit{res} \Longrightarrow A\ \mathit{res}$ is also evident, but is not of interest here since we never consider the judgment A *res* in the succedent of a sequent.

We also need a rule that allows a factory to produce a resource. This rule is called copy and sometimes refered to as *dereliction*.

$$\frac{(\Gamma, v{:}A\,\mathit{fact}); (\Delta, w{:}A\ \mathit{res}) \Longrightarrow C\ \mathit{goal}}{(\Gamma, v{:}A\,\mathit{fact}); \Delta \Longrightarrow C\ \mathit{goal}}\ \mathsf{copy}_v$$

Note how this is different from the unrestricted hypothesis rule in natural deduction. Factories are directly related to resources and only indirectly to goals.

The remaining rules are divided into *right* and *left* rules, which correspond to the *introduction* and *elimination* rules of natural deduction, respectively. The right rules apply to the goal, while the left rules apply to resources. In the following, we adhere to common practice and omit labels on hypotheses and consequently also on the justifications of the inference rules. The reader should keep in mind, however, that this is just a short-hand, and that there are, for example, two *different* derivations of $(A, A); \cdot \Longrightarrow A$, one using the first copy of A and one using the second.

Hypotheses.

$$\frac{}{\Gamma; A \Longrightarrow A}\,\mathsf{init} \qquad \frac{(\Gamma, A); (\Delta, A) \Longrightarrow C}{(\Gamma, A); \Delta \Longrightarrow C}\,\mathsf{copy}$$

Multiplicative Connectives.

$$\frac{\Gamma; \Delta, A \Longrightarrow B}{\Gamma; \Delta \Longrightarrow A \multimap B}\,{\multimap}\mathrm{R} \qquad \frac{\Gamma; \Delta_1 \Longrightarrow A \qquad \Gamma; \Delta_2, B \Longrightarrow C}{\Gamma; \Delta_1, \Delta_2, A \multimap B \Longrightarrow C}\,{\multimap}\mathrm{L}$$

$$\frac{\Gamma; \Delta_1 \Longrightarrow A \qquad \Gamma; \Delta_2 \Longrightarrow B}{\Gamma; \Delta_1, \Delta_2 \Longrightarrow A \otimes B}\,{\otimes}\mathrm{R} \qquad \frac{\Gamma; \Delta, A, B \Longrightarrow C}{\Gamma; \Delta, A \otimes B \Longrightarrow C}\,{\otimes}\mathrm{L}$$

$$\frac{}{\Gamma; \cdot \Longrightarrow \mathbf{1}}\,\mathbf{1}\mathrm{R} \qquad \frac{\Gamma; \Delta \Longrightarrow C}{\Gamma; \Delta, \mathbf{1} \Longrightarrow C}\,\mathbf{1}\mathrm{L}$$

Additive Connectives.

$$\frac{\Gamma; \Delta \Longrightarrow A \qquad \Gamma; \Delta \Longrightarrow B}{\Gamma; \Delta \Longrightarrow A \& B}\,\&\mathrm{R} \qquad \begin{array}{c} \dfrac{\Gamma; \Delta, A \Longrightarrow C}{\Gamma; \Delta, A \& B \Longrightarrow C}\,\&\mathrm{L}_1 \\[2ex] \dfrac{\Gamma; \Delta, B \Longrightarrow C}{\Gamma; \Delta, A \& B \Longrightarrow C}\,\&\mathrm{L}_2 \end{array}$$

$$\frac{}{\Gamma; \Delta \Longrightarrow \top}\,\top\mathrm{R} \qquad \textit{No } \top \textit{ left rule}$$

$$\begin{array}{c} \dfrac{\Gamma; \Delta \Longrightarrow A}{\Gamma; \Delta \Longrightarrow A \oplus B}\,\oplus\mathrm{R}_1 \\[2ex] \dfrac{\Gamma; \Delta \Longrightarrow B}{\Gamma; \Delta \Longrightarrow A \oplus B}\,\oplus\mathrm{R}_2 \end{array} \qquad \frac{\Gamma; \Delta, A \Longrightarrow C \qquad \Gamma; \Delta, B \Longrightarrow C}{\Gamma; \Delta, A \oplus B \Longrightarrow C}\,\oplus\mathrm{L}$$

$$\textit{No } \mathbf{0} \textit{ right rule} \qquad \frac{}{\Gamma; \Delta, \mathbf{0} \Longrightarrow C}\,\mathbf{0}\mathrm{L}$$

Quantifiers.

$$\frac{\Gamma; \Delta \Longrightarrow [a/x]A}{\Gamma; \Delta \Longrightarrow \forall x.\, A}\,\forall\mathrm{R}^a \qquad \frac{\Gamma; \Delta, [t/x]A \Longrightarrow C}{\Gamma; \Delta, \forall x.\, A \Longrightarrow C}\,\forall\mathrm{L}$$

$$\frac{\Gamma; \Delta \Longrightarrow [t/x]A}{\Gamma; \Delta \Longrightarrow \exists x.\, A}\,\exists\mathrm{R} \qquad \frac{\Gamma; \Delta, [a/x]A \Longrightarrow C}{\Gamma; \Delta, \exists x.\, A \Longrightarrow C}\,\exists\mathrm{L}^a$$

Exponentials.

$$\frac{(\Gamma, A); \Delta \Longrightarrow B}{\Gamma; \Delta \Longrightarrow A \supset B}\,\supset\mathrm{R} \qquad \frac{\Gamma; \cdot \Longrightarrow A \qquad \Gamma; \Delta, B \Longrightarrow C}{\Gamma; \Delta, A \supset B \Longrightarrow C}\,\supset\mathrm{L}$$

$$\frac{\Gamma; \cdot \Longrightarrow A}{\Gamma; \cdot \Longrightarrow\, !A}\,!\mathrm{R} \qquad \frac{(\Gamma, A); \Delta \Longrightarrow C}{\Gamma; (\Delta, !A) \Longrightarrow C}\,!\mathrm{L}$$

To obtain a normal deduction from a sequent derivation we map instances of right rules to corresponding introduction rules. Left rules have to be turned "upside-down", since the elimination rule corresponding to a left rule works in the opposite direction. This reverse of direction is captured in the proof of the following theorem by appeals to the substitution property: we extend a natural deduction at a leaf by substituting a one-step deduction for the use of a hypothesis. Note that the terse statement of this theorem (and also of the completeness theorem below) hide the fact that the judgments forming the assumptions Γ and Δ are different in the sequent calculus and natural deduction.

Theorem 3.1 (Soundness of Sequent Derivations)
If $\Gamma; \Delta \Longrightarrow A$ *then* $\Gamma; \Delta \vdash A \uparrow$.

Proof: By induction on the structure of the derivation $\mathcal{S}$ of $\Gamma; \Delta \Longrightarrow A$. Initial sequents are translated to the $\downarrow\uparrow$ coercion, and use of an unrestricted hypothesis follows by a substitution principle (Lemma 2.2). For right rules we apply the corresponding introduction rules. For left rules we either directly construct a derivation of the conclusion after an appeal to the induction hypothesis ($\otimes$L, **1**L, $\otimes$L, **0**L, $\exists$L, !L) or we appeal to a substitution principle of atomic natural deductions for hypotheses ($\multimap$L, &L_1, &L_2, $\forall$L, $\supset$L). We should four cases in detail. We write out explicit judgments in some cases for the sake of clarity.

Case: $\mathcal{S}$ ends in init.

$$\mathcal{S} = \frac{}{\Gamma, A\ \mathit{res} \Longrightarrow A\ \mathit{goal}}\,\mathsf{init}$$

$\Gamma; A\downarrow \vdash A\downarrow$	Linear hypothesis
$\Gamma; A\downarrow \vdash A\uparrow$	By rule $\downarrow\uparrow$

Case: $\mathcal{S}$ ends in copy.

$$\mathcal{S} = \dfrac{\begin{matrix}\mathcal{S}_1\\ (\Gamma', B\,\mathit{fact}); (\Delta, B\,\mathit{res}) \Longrightarrow A\ \mathit{goal}\end{matrix}}{(\Gamma', B\,\mathit{fact}); \Delta \Longrightarrow A\ \mathit{goal}}\ \mathsf{copy}$$

$(\Gamma', B\,\mathit{fact}); (\Delta, B\,\mathit{res}) \Longrightarrow A\ \mathit{goal}$	Subderivation
$(\Gamma', B\Downarrow); (\Delta, B\downarrow) \vdash A\uparrow$	By i.h. on $\mathcal{S}_1$
$(\Gamma', B\Downarrow); \cdot \vdash B\downarrow$	Unrestricted hypothesis
$(\Gamma', B\Downarrow); \Delta \vdash A\uparrow$	By substitution property (2.2)

Case: $\mathcal{S}$ ends in $\otimes$R.

$$\mathcal{S} = \dfrac{\begin{matrix}\mathcal{S}_1\\ \Gamma; \Delta_1 \Longrightarrow A_1\end{matrix} \qquad \begin{matrix}\mathcal{S}_2\\ \Gamma; \Delta_2 \Longrightarrow A_2\end{matrix}}{\Gamma; (\Delta_1, \Delta_2) \Longrightarrow A_1 \otimes A_2}\ \otimes\mathrm{R}$$

$\Gamma; \Delta_1 \vdash A_1\uparrow$	By i.h. on $\mathcal{S}_1$
$\Gamma; \Delta_2 \vdash A_2\uparrow$	By i.h. on $\mathcal{S}_2$
$\Gamma; (\Delta_1, \Delta_2) \vdash A_1 \otimes A_2\uparrow$	By rule $\otimes$I

Case: $\mathcal{S}$ end in $\&\mathrm{L}_1$

$$\mathcal{S} = \dfrac{\begin{matrix}\mathcal{S}_1\\ \Gamma; \Delta, B_1 \Longrightarrow A\end{matrix}}{\Gamma; \Delta, B_1 \& B_2 \Longrightarrow A}\ \&\mathrm{L}_1$$

$\Gamma; \Delta, B_1 \vdash A\uparrow$	By i.h. on $\mathcal{S}_1$
$\Gamma; B_1 \& B_2 \vdash B_1 \& B_2\downarrow$	Linear hypothesis
$\Gamma; B_1 \& B_2 \vdash B_1\downarrow$	By rule $\&\mathrm{E_L}$
$\Gamma; \Delta, B_1 \& B_2 \vdash A\uparrow$	By substitution property (2.2)

□

The completeness theorem reverses the translation from above. In this case we have to generalize the induction hypothesis so we can proceed when we encounter a coercion from atomic to normal derivations. It takes some experience to find the generalization we give below. Fortunately, the rest of the proof is then straightforward.

Theorem 3.2 (Completeness of Sequent Derivations)

1. *If* $\Gamma; \Delta \vdash A\uparrow$ *then there is a sequent derivation of* $\Gamma; \Delta \Longrightarrow A$*, and*

2. *if* $\Gamma; \Delta \vdash A\downarrow$ *then for any formula* C *and derivation of* $\Gamma; \Delta', A \Longrightarrow C$ *there is a derivation of* $\Gamma; (\Delta', \Delta) \Longrightarrow C$.

Proof: By simultaneous induction on the structure of the derivations of $\mathcal{N}$ of $\Gamma; \Delta \vdash A \uparrow$ and $\mathcal{A}$ of $\Gamma; \Delta \vdash A \downarrow$. We show a few representative cases.

Case: $\mathcal{N}$ ends in $\downarrow\uparrow$.

$$\mathcal{N} = \cfrac{\begin{array}{c}\mathcal{A}\\ \Gamma; \Delta \vdash A \downarrow\end{array}}{\Gamma; \Delta \vdash A \uparrow}\downarrow\uparrow$$

$\Gamma; \Delta', A \Longrightarrow C$ implies $\Gamma; \Delta', \Delta \Longrightarrow C$ for any Δ' and C	By i.h. on $\mathcal{A}$
$\Gamma; A \Longrightarrow A$	By rule init
$\Gamma; \Delta \Longrightarrow A$	From i.h. using $\Delta' = \cdot$ and $C = A$

Case: $\mathcal{A}$ ends in $\&\mathrm{E_L}$.

$$\mathcal{A} = \cfrac{\begin{array}{c}\mathcal{A}_1\\ \Gamma; \Delta \vdash A \& B \downarrow\end{array}}{\Gamma; \Delta \vdash A \downarrow}\&\mathrm{E_L}$$

$\Gamma; \Delta', A \& B \Longrightarrow C$ implies $\Gamma; \Delta', \Delta \Longrightarrow C$ for any Δ' and C	By i.h. on $\mathcal{A}_1$
$\Gamma; \Delta', A \Longrightarrow C$ for some Δ' and C	New assumption
$\Gamma; \Delta', A \& B \Longrightarrow C$	By rule $\&\mathrm{E_L}$
$\Gamma; \Delta', \Delta \Longrightarrow C$	From i.h. using $\Delta' = \Delta'$ and $C = C$

Case: $\mathcal{A}$ is an appeal to a linear hypothesis.

$$\mathcal{A} = \cfrac{}{\Gamma; w{:}A \downarrow \vdash A \downarrow}w$$

$\Gamma; \Delta', A \Longrightarrow C$ for some Δ' and C	New assumption
$(\Delta', \Delta) = (\Delta', A)$ since $\Delta = A$	This case

Case: $\mathcal{A}$ is an appeal to an unrestricted hypothesis.

$$\mathcal{A} = \cfrac{}{(\Gamma', v{:}A \downarrow\!\downarrow); \cdot \vdash A \downarrow}v$$

$(\Gamma', A); \Delta', A \Longrightarrow C$ for some Δ' and C	New assumption
$(\Gamma', A); \Delta' \Longrightarrow C$	By rule copy
$\Gamma = (\Gamma', A)$ and $\Delta = \cdot$	This case

$\square$

3.2 Another Example: Petri Nets

In this section we show how to represent Petri nets in linear logic. This example is due to Martì-Oliet and Meseguer [MOM91], but has been treated several times in the literature.

A *Petri net* is defined by a collection of *places*, *transitions*, *arcs*, and *tokens*. Every transition has input arcs and output arcs that connect it to places. The system evolves by changing the tokens in various places according to the following rules.

1. A transition is enabled if every place connected to it by an input arc contains at least one token.

2. We non-deterministically select one of the enabled transitions in a net to fire.

3. A transition fires by removing one token from each input place and adding one token to each output place of the transition.

Slightly more generally, an arc may have a *weight* n. For an input arc this means there must be at least n tokens on the place to enable a transition. When the transition fires, n tokens are removed from the token at the beginning of an arc with weight n. For an output arc this means that n new tokens will be added to the place at its end. By default, an arc with no listed weight has weight one. There are other variations and generalizations of Petri nets, but we will not consider them here. Figure 3.1 displays some typical Petri net structures.

It is quite easy to represent a Petri net in linear logic. The idea is that the fixed topology of the net is represented as a collection of unrestricted propositions Γ. The current state of the net as given by the tokens in the net is represent as collection of resources Δ. We can reach state Δ_1 from state Δ_0 iff $\Gamma; \cdot \vdash (\bigotimes \Delta_0) \multimap (\bigotimes \Delta_1)$. That is, provability will correspond precisely to reachability in the Petri net. We formulate this below in a slightly differently, using the sequent calculus as a tool.

To accomplish this, we represent every place by an atomic predicate. If there are k tokens on place p, we add k copies of p into the representation of the state Δ. For every transition we add a rule $p_1 \otimes \cdots \otimes p_m \multimap q_1 \otimes \cdots \otimes q_n$ to Γ, where $p_1, \ldots, p_m$ are the places at the beginning of the input arcs and $q_1 \otimes \cdots \otimes q_n$ are the places at the end of the output arcs. If an arc has multiplicity k, we simply add k copies of p to either the antecedent or the succedent of the corresponding linear implication representing the transition. As an example, consider the following Petri net (used in [Cer95]).

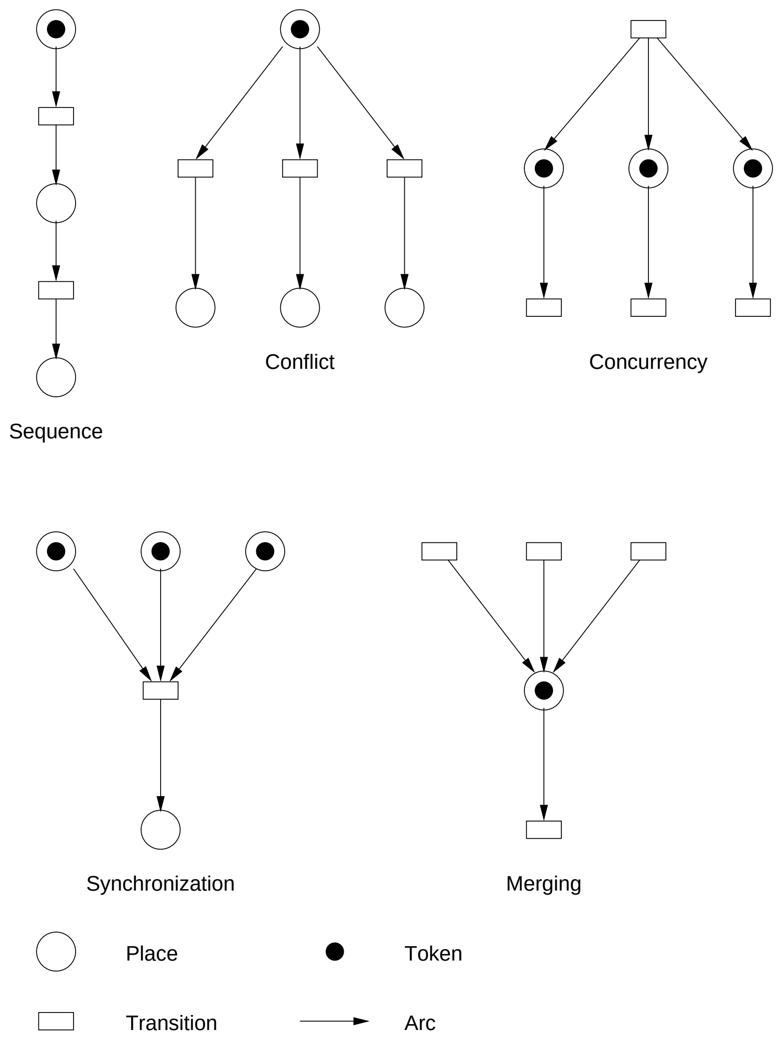
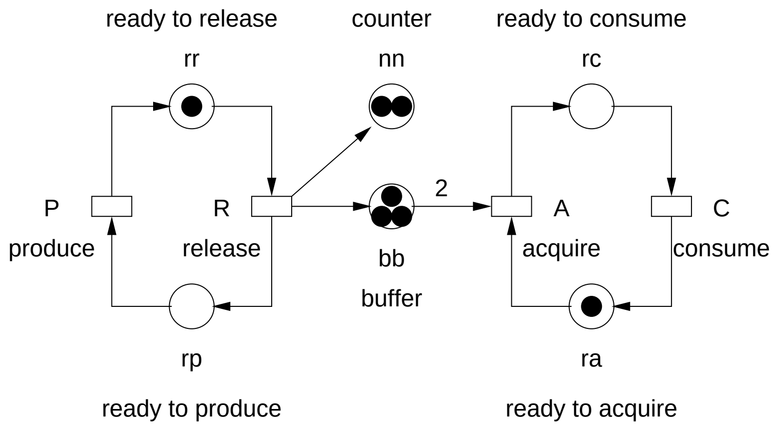


Figure 3.1: Some Petri Net Structures



Note that arc form the buffer to the acquire transition has weight two, so two tokens in the buffer are converted to one token to be in the place ready to consume.

The representation of this Petri net consists of the following unrestricted rule in Γ and the initial state in Δ_0.

$$\begin{array}{rcl} \Gamma & = & P : \mathsf{rp} \multimap \mathsf{rr} \\ & & R : \mathsf{rr} \multimap \mathsf{rp} \otimes \mathsf{nn} \otimes \mathsf{bb} \\ & & A : \mathsf{bb} \otimes \mathsf{bb} \otimes \mathsf{ra} \multimap \mathsf{rc} \\ & & C : \mathsf{rc} \multimap \mathsf{ra} \\ \Delta_0 & = & \mathsf{rr}, \mathsf{nn}, \mathsf{nn}, \mathsf{bb}, \mathsf{bb}, \mathsf{bb}, \mathsf{ra} \end{array}$$

Informally, it is quite easy to understand that the propositions above represent the given Petri nets. We now consider a slightly different from of the adequacy theorem in order exploit the sequent calculus

Adequacy for Encoding of Petri Nets.

Assume we are given a Petri net with places $P = \{p_1, \ldots, p_n\}$, transitions $T = \{t_1, \ldots, t_m\}$. We represent the transitions as unrestricted assumptions Γ as sketched above, and a token assignment as a collection of linear hypotheses $\Delta = (q_1, \ldots, q_k)$ where q_j are places, possibly containing repetitions. Then the marking Δ_1 is reachable from marking Δ_0 if and only if

$$\begin{array}{c} \Gamma; \Delta_1 \Longrightarrow C \\ \vdots \\ \Gamma; \Delta_0 \Longrightarrow C \end{array}$$

for an arbitrary proposition C.

Considered bottom up, this claims that, for any C, we can reduce the problem of proving $\Gamma; \Delta_0 \vdash C$ to the problem of proving $\Gamma; \Delta_1 \vdash C$.

3.3 Deductions with Lemmas

One common way to find or formulate a proof is to introduce a lemma. In the sequent calculus, the introduction and use of a lemma during proof search is modelled by the rules of cut, cut for lemmas used as linear hypotheses, and $\mathsf{cut!}$ for lemmas used as factories or resources. The corresponding rule for intuitionistic logic is due to Gentzen [Gen35]. We write $\Gamma; \Delta \stackrel{+}{\Longrightarrow} A$ for the judgment that A can be derived with the rules from before, plus one of the two cut rules below.

$$\frac{\Gamma; \Delta \stackrel{+}{\Longrightarrow} A \quad \Gamma; (\Delta', A) \stackrel{+}{\Longrightarrow} C}{\Gamma; \Delta, \Delta' \stackrel{+}{\Longrightarrow} C}\,\mathsf{cut} \qquad \frac{\Gamma; \cdot \stackrel{+}{\Longrightarrow} A \quad (\Gamma, A); \Delta' \stackrel{+}{\Longrightarrow} C}{\Gamma; \Delta' \stackrel{+}{\Longrightarrow} C}\,\mathsf{cut!}$$

Note that the linear context in the left premise of the $\mathsf{cut!}$ rule must be empty, because the new hypothesis A in the right premise is unrestricted in its use.

From the judgmental point of view, the first cut rule corresponds to the inverse of the init rule. Ignoring extraneous hypotheses, the init rule states $A\ \mathit{res} \Longrightarrow A\ \mathit{goal}$. To go the opposity way means that we are allowed to assume $A\ \mathit{res}$ if we have shown $A\ \mathit{goal}$. This is exactly what the cut rule expresses. The $\mathsf{cut!}$ expresses that if we can achieve a goal A without using any linear resources, we can manufacture as many copies of the resource A as we like.

On the side of natural deduction, these rules correspond to substitution principles. They can be related to normal and atomic derivations only if we allow an additional coercion from normal to atomic derivations. This is because the left premise corresponds to a derivation of $\Gamma; \Delta \vdash A\uparrow$ which can be substituted into a derivation of $\Gamma; \Delta', A\downarrow \vdash C\uparrow$ only if we have this additional coercion. Of course, the resulting deductions are no longer normal in the sense we defined before, so we write $\Gamma; \Delta \vdash^{+} A\downarrow$ and $\Gamma; \Delta \vdash^{+} A\uparrow$. These judgments are defined with the same rules as $\Gamma; \Delta \vdash A\uparrow$ and $\Gamma; \Delta \vdash A\downarrow$, plus the following coercion.

$$\frac{\Gamma; \Delta \vdash^{+} A\uparrow}{\Gamma; \Delta \vdash^{+} A\downarrow}\,\uparrow\downarrow$$

It is now easy to prove that arbitrary natural deductions can be annotated with $\uparrow$ and $\downarrow$, since we can arbitrarily coerce back and forth between the two judgments.

Theorem 3.3 *If* $\Gamma; \Delta \vdash A$ *then* $\Gamma; \Delta \vdash^{+} A\uparrow$ *and* $\Gamma; \Delta \vdash^{+} A\downarrow$

Proof: By induction on the structure of $\mathcal{D} :: (\Gamma; \Delta \vdash A)$. □

Theorem 3.4

1. *If* $\Gamma; \Delta \vdash^{+} A\uparrow$ *then* $\Gamma; \Delta \vdash A$.
2. *If* $\Gamma; \Delta \vdash^{+} A\downarrow$ *then* $\Gamma; \Delta \vdash A$.

Proof: My mutual induction on $\mathcal{N} :: (\Gamma; \Delta \vdash^+ A\uparrow)$ and $\mathcal{A} :: (\Gamma; \Delta \vdash^+ A\downarrow)$. $\square$

It is also easy to relate the cut rules to the new coercions (and thereby to natural deductions), plus four substitution principles.

Property 3.5 (Substitution)

1. *If* $\Gamma; \Delta \vdash^+ A\downarrow$ *and* $\Gamma; (\Delta', u{:}A\downarrow) \vdash^+ C\uparrow$ *then* $\Gamma; (\Delta, \Delta') \vdash^+ C\uparrow$.
2. *If* $\Gamma; \Delta \vdash^+ A\downarrow$ *and* $\Gamma; (\Delta', u{:}A\downarrow) \vdash^+ C\downarrow$ *then* $\Gamma; (\Delta, \Delta') \vdash^+ C\downarrow$.
3. *If* $\Gamma; \cdot \vdash^+ A\downarrow$ *and* $(\Gamma, v{:}A\downarrow\!\!\downarrow); \Delta' \vdash^+ C\uparrow$ *then* $\Gamma; \Delta' \vdash^+ C\uparrow$.
4. *If* $\Gamma; \cdot \vdash^+ A\downarrow$ *and* $(\Gamma, v{:}A\downarrow\!\!\downarrow); \Delta' \vdash^+ C\downarrow$ *then* $\Gamma; \Delta' \vdash^+ C\downarrow$.

Proof: By mutual induction on the structure of the given derivations. $\square$

We can now extend Theorems 3.1 and 3.2 to relate sequent derivations with cut to natural deductions with explicit lemmas.

Theorem 3.6 (Soundness of Sequent Derivations with Cut)
If $\Gamma; \Delta \overset{+}{\Longrightarrow} A$ *then* $\Gamma; \Delta \vdash^+ A\uparrow$.

Proof: As in Theorem 3.1 by induction on the structure of the derivation of $\Gamma; \Delta \overset{+}{\Longrightarrow} A$. An inference with one of the new rules cut or $\mathsf{cut!}$ is translated into an application of the $\uparrow\downarrow$ coercion followed by an appeal to one of the substitution principles in Property 3.5. $\square$

Theorem 3.7 (Completeness of Sequent Derivations with Cut)

1. *If* $\Gamma; \Delta \vdash^+ A\uparrow$ *then there is a sequent derivation of* $\Gamma; \Delta \overset{+}{\Longrightarrow} A$, *and*
2. *if* $\Gamma; \Delta \vdash^+ A\downarrow$ *then for any formula* C *and derivation of* $\Gamma; (\Delta', A) \overset{+}{\Longrightarrow} C$ *there is a derivation of* $\Gamma; (\Delta', \Delta) \overset{+}{\Longrightarrow} C$.

Proof: As in the proof of Theorem 3.2 by induction on the structure of the given derivations. In the new case of the $\uparrow\downarrow$ coercion, we use the rule of cut. The other new rule, $\mathsf{cut!}$, is not needed for this proof, but is necessary for the proof of admissibility of cut in the next section. We show the new case.

Case: $\mathcal{A}$ ends in $\uparrow\downarrow$.

$$\mathcal{A} = \cfrac{\begin{array}{c}\mathcal{N}\\ \Gamma; \Delta \vdash^+ A\uparrow\end{array}}{\Gamma; \Delta \vdash^+ A\downarrow}\uparrow\downarrow$$

$\Gamma; \Delta \overset{+}{\Longrightarrow} A$	By i.h. on $\mathcal{N}$
$\Gamma; \Delta', A \overset{+}{\Longrightarrow} C$ for some Δ' and C	New assumption
$\Gamma; \Delta, \Delta' \overset{+}{\Longrightarrow} C$	By rule cut

$\square$

3.4 Cut Elimination

We viewed the sequent calculus as a calculus of proof search for natural deduction. The proofs of the soundness theorems 3.2 and 3.7 provide ways to translate cut-free sequent derivations into normal natural deductions, and sequent derivations with cut into arbitrary natural deductions.

This section is devoted to showing that the two rules of cut are redundant in the sense that any derivation in the sequent calculus which makes use of the rules of cut can be translated to one that does not. Taken together with the soundness and completeness theorems for the sequent calculi with and without cut, this has many important consequences.

First of all, a proof search procedure which looks only for cut-free sequent derivations will be complete: any derivable proposition can be proven this way. When the cut rule

$$\frac{\Gamma; \Delta \stackrel{+}{\Longrightarrow} A \qquad \Gamma; (\Delta', A) \stackrel{+}{\Longrightarrow} C}{\Gamma; \Delta', \Delta \stackrel{+}{\Longrightarrow} C}\,\mathsf{cut}$$

is viewed in the bottom-up direction the way it would be used during proof search, it introduces a new and arbitrary proposition A. Clearly, this introduces a great amount of non-determinism into the search. The cut elimination theorem now tells us that we never need to use this rule. All the remaining rules have the property that the premises contain only instances of propositions in the conclusion, or parts thereof. This latter property is often called the *subformula property*.

Secondly, it is easy to see that the logic is *consistent*, that is, not every proposition is provable. In particular, the sequent $\cdot;\cdot \Longrightarrow \mathbf{0}$ does not have a cut-free derivation, because there is simply no rule which could be applied to infer it! This property clearly fails in the presence of cut: it is *prima facie* quite possible that the sequent $\cdot;\cdot \stackrel{+}{\Longrightarrow} \mathbf{0}$ is the conclusion of the cut rule.

Along the same lines, we can show that a number of propositions are *not derivable* in the sequent calculus and therefore not true as defined by the natural deduction rules. Examples of this kind are given at the end of this section.

We prove cut elimination by showing that the two cut rules are *admissible rules of inference* in the sequent calculus without cut. An inference rule is admissible if whenever we can find derivations for its premises we can find a derivation of its conclusion. This should be distinguished from a *derived rule of inference* which requires a direct derivation of the conclusion from the premises. We can also think of a derived rule as an evident hypothetical judgment where the premises are (unrestricted) hypotheses.

Derived rules of inference have the important property that they remain evident under any extension of the logic. An admissible rule, on the other hand, represents a global property of the deductive system under consideration and may well fail when the system is extended. Of course, every derived rule is also admissible.

Theorem 3.8 (Admissibility of Cut)

1. *If* $\Gamma; \Delta \Longrightarrow A$ *and* $\Gamma; (\Delta', A) \Longrightarrow C$ *then* $\Gamma; (\Delta, \Delta') \Longrightarrow C$.
2. *If* $\Gamma; \cdot \Longrightarrow A$ *and* $(\Gamma, A); \Delta' \Longrightarrow C$ *then* $\Gamma; \Delta' \Longrightarrow C$.

Proof: By nested inductions on the structure of the cut formula A and the given derivations, where induction hypothesis (1) has priority over (2). To state this more precisely, we refer to the given derivations as $\mathcal{D} :: (\Gamma; \Delta \Longrightarrow A)$, $\mathcal{D}' :: (\Gamma; \cdot \Longrightarrow A)$, $\mathcal{E} :: (\Gamma; (\Delta, A) \Longrightarrow C)$, and $\mathcal{E}' :: ((\Gamma, A); \Delta' \vdash C)$. Then we may appeal to the induction hypothesis whenever

a. the cut formula A is strictly smaller, or

b. the cut formula A remains the same, but we appeal to induction hypothesis (1) in the proof of (2) (but when we appeal to (2) in the proof of (1) the cut formula must be strictly smaller), or

c. the cut formula A and the derivation $\mathcal{E}$ remain the same, but the derivation $\mathcal{D}$ becomes smaller, or

d. the cut formula A and the derivation $\mathcal{D}$ remain the same, but the derivation $\mathcal{E}$ or $\mathcal{E}'$ becomes smaller.

Here, we consider a formula smaller it is an immediate subformula, where $[t/x]A$ is considered a subformula of $\forall x.\ A$, since it contains fewer quantifiers and logical connectives. A derivation is smaller if it is an immediate subderivation, where we allow weakening by additional unrestricted hypothesis in one case (which does not affect the structure of the derivation).

The cases we have to consider fall into 5 classes:

Initial Cuts: One of the two premises is an initial sequent. In these cases the cut can be eliminated directly.

Principal Cuts: The cut formula A was just inferred by a right rule in $\mathcal{D}$ and by a left rule in $\mathcal{E}$. In these cases we appeal to the induction hypothesis (possibly several times) on smaller cut formulas (item (a) above).

Copy Cut: The cases for the $\mathsf{cut}!$ rule are treated as right commutative cuts (see below), except for the rule of dereliction which requires an appeal to induction hypothesis (1) with the same cut formula (item (b) above).

Left Commutative Cuts: The cut formula A is a side formula of the last inference in $\mathcal{D}$. In these cases we may appeal to the induction hypotheses with the same cut formula, but smaller derivation $\mathcal{D}$ (item (c) above).

Right Commutative Cuts: The cut formula A is a side formula of the last inference in $\mathcal{E}$. In these cases we may appeal to the induction hypotheses with the same cut formula, but smaller derivation $\mathcal{E}$ or $\mathcal{E}'$ (item (d) above).

We show one case from each category.

Case: Initial cut where $\mathcal{D}$ is initial and $\mathcal{E}$ is arbitrary.

$$\mathcal{D} = \frac{}{\Gamma; A \Longrightarrow A}\,\mathsf{init}$$

$\Delta = A$	This case
$\Gamma; \Delta', A \Longrightarrow C$	Derivation $\mathcal{E}$

Case: Principal cut, where $\mathcal{D}$ ends in $\otimes$R and $\mathcal{E}$ end in $\otimes$L.

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ \Gamma; \Delta_1 \vdash A_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Gamma; \Delta_2 \vdash A_2\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash A_1 \otimes A_2}\,\otimes\mathrm{R}$$

and

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1\\ \Gamma; \Delta', A_1, A_2 \Longrightarrow C\end{array}}{\Gamma; \Delta', A_1 \otimes A_2 \Longrightarrow C}\,\otimes\mathrm{L}$$

$\Gamma; \Delta', \Delta_1, A_2 \Longrightarrow C$	By i.h. on A_1, $\mathcal{D}_1$ and $\mathcal{E}_1$
$\Gamma; \Delta', \Delta_1, \Delta_2 \Longrightarrow C$	By i.h. on A_2, $\mathcal{D}_2$ and above

Case: Copy cut, where $\mathcal{D}$ is arbitrary and $\mathcal{E}$ ends in copy.

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1\\ (\Gamma', A); (\Delta', A) \Longrightarrow C\end{array}}{(\Gamma', A); \Delta' \Longrightarrow C}\,\mathsf{copy}$$

$\Gamma; \cdot \Longrightarrow A$	Derivation $\mathcal{D}$
$\Gamma'; (\Delta', A) \Longrightarrow C$	By i.h.(2) on A, $\mathcal{D}$ and $\mathcal{E}_1$
$\Gamma'; \Delta' \Longrightarrow C$	By i.h.(1) on A, $\mathcal{D}$ and above

Case: Left commutative cut, where $\mathcal{D}$ ends in $\&\mathrm{L}_1$ and $\mathcal{E}$ is arbitrary.

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ \Gamma; \Delta_1, B_1 \Longrightarrow A\end{array}}{\Gamma; \Delta_1, B_1 \& B_2 \Longrightarrow A}\,\&\mathrm{L}_1$$

$\Gamma; \Delta', A \Longrightarrow C$	Derivation $\mathcal{E}$
$\Gamma; \Delta', \Delta_1, B_1 \Longrightarrow C$	By i.h. on A, $\mathcal{D}_1$ and $\mathcal{E}$
$\Gamma; \Delta', \Delta_1, B_1 \& B_2 \Longrightarrow C$	By rule $\&\mathrm{L}_1$

Case: Right commutative cut, where $\mathcal{D}$ is arbitrary and $\mathcal{E}$ ends in $\oplus\mathrm{R}_1$.

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1\\ \Gamma; \Delta, A \Longrightarrow C_1\end{array}}{\Gamma; \Delta, A \Longrightarrow C_1 \oplus C_2}\,\oplus\mathrm{R}_1$$

$\Gamma; \Delta \Longrightarrow A$	Derivation $\mathcal{D}$
$\Gamma; \Delta, \Delta' \Longrightarrow C_1$	By i.h. on A, $\mathcal{D}$ and $\mathcal{E}_1$
$\Gamma; \Delta \Longrightarrow C_1 \oplus C_2$	By rule $\oplus R_1$

□

Using the admissibility of cut, the cut elimination theorem follows by a simple structural induction.

Theorem 3.9 (Cut Elimination)
If $\Gamma; \Delta \stackrel{+}{\Longrightarrow} C$ *then* $\Gamma; \Delta \Longrightarrow C$.

Proof: By induction on the structure of $\mathcal{D} :: (\Gamma; \Delta \stackrel{+}{\Longrightarrow} C)$. In each case except cut or $\mathsf{cut}!$ we simply appeal to the induction hypothesis on the derivations of the premises and use the corresponding rule in the cut-free sequent calculus. For the cut and $\mathsf{cut}!$ rules we appeal to the induction hypothesis and then admissibility of cut (Theorem 3.8) on the resulting derivations. □

3.5 Consequences of Cut Elimination

The first and most important consequence of cut elimination is that every natural deduction can be translated to a normal natural deduction. The necessary construction is implicit in the proofs of the soundness and completeness theorems for sequent calculi and the proofs of admissibility of cut and cut elimination. In Chapter 6 we will see a much more direct, but in other respects more complicated proof.

Theorem 3.10 (Normalization for Natural Deductions)
If $\Gamma; \Delta \vdash A$ *then* $\Gamma; \Delta \vdash A\uparrow$.

Proof: Directly, using theorems from this chapter.

$\Gamma; \Delta \vdash A$	Assumption
$\Gamma; \Delta \vdash^{+} A$	By Theorem 3.3
$\Gamma; \Delta \stackrel{+}{\Longrightarrow} A$	By completeness of sequent derivations with cut (Theorem 3.7)
$\Gamma; \Delta \Longrightarrow A$	By cut elimination (Theorem 3.9)
$\Gamma; \Delta \vdash A\uparrow$	By soundness of cut-free sequent derivations (Theorem 3.1)

□

As a second consequence, we see that linear logic is *consistent*: not every proposition can be proved. A proof of consistency for both intuitionistic and classical logic was Gentzen's original motivation for the development of the sequent calculus and his proof of cut elimination.

Theorem 3.11 (Consistency of Intuitionistic Linear Logic)
$\cdot;\cdot \vdash \mathbf{0}$ *true is not derivable.*

Proof: If the judgment were derivable, by Theorems 3.3, 3.7, and 3.9, there must be a cut-free sequent derivation of $\cdot;\cdot \Longrightarrow \mathbf{0}$. But there is no rule with which we could infer this sequent (there is no right rule for $\mathbf{0}$), and so it cannot be derivable. □

A third consequence is called the *disjunction property.* Note that in ordinary classical logic this property fails.

Theorem 3.12 (Disjunction Property for Intuitionistic Linear Logic) *If* $\cdot;\cdot \vdash A \oplus B$ *true then either* $\cdot;\cdot \vdash A$ *true or* $\cdot;\cdot \vdash B$ *true.*

Proof: Assume $\cdot;\cdot \vdash A \oplus B$ *true*. Then, by completeness of the cut-free sequent calculus, $\cdot;\cdot \Longrightarrow A \oplus B$. But there are only two rules that end with this judgment: $\oplus R_1$ and $\oplus R_2$. Hence either $\cdot;\cdot \Longrightarrow A$ or $\cdot;\cdot \Longrightarrow B$. Therefore, by soundness of the sequent calculus, $\cdot;\cdot \vdash A$ *true* or $\cdot;\cdot \vdash B$ *true* □

Note that these theorems are just special cases, and many other properties of the connectives follow from normalization and cut elimination.

As another kind of example, we can show that various propositions are *not* theorems of linear logic. Consider

$$\cdot; A \multimap (B \otimes C) \vdash (A \multimap B) \otimes (A \multimap C)$$

Intuitively, this should clearly not hold for arbitrary A, B, and C (although it could be true for some specific ones). But if we know the completeness of the cut-free sequent calculus this is easy to show. Consider

$$\cdot; A \multimap (B \otimes C) \Longrightarrow (A \multimap B) \otimes (A \multimap C).$$

There are only two possible rules that could have been used to deduce this conclusion, $\multimap R$ and $\otimes R$.

In case the last rule is $\multimap R$, one of the premises will be

$$\cdot;\cdot \Longrightarrow A$$

which is not provable for arbitrary A. In case the last rule is $\otimes R$, the linear hypothesis must be propagated to the left or right premise. Assume it goes to the left (the other case is symmetric). Then the right premise must be

$$\cdot;\cdot \Longrightarrow A \multimap C$$

which could only be infered by $\multimap R$, which leaves

$$\cdot; A \Longrightarrow C.$$

Again, unless we know more about A and C no rule applies. Hence the judgment above has no proof.

3.6 Another Example: The π-Calculus

The π-calculus was designed by Milner as a foundational calculus to investigate properties of communicating and mobile systems [Mil99]. The first formulation below differs slightly from Milner's in the details of specification, but one can also give a completely faithful representation as shown in our second version.[1]

The basic syntactic categories in the π-calculus are *names*, *actions*, and *processes*. Names are simply written as variables x, y or a. They serve simultaneously as communication channels and the data that is transmitted along the channels. They constitute the only primitive data objects in the π-calculus, which makes it somewhat tedious to write non-trivial examples. In this sense it is similar to Church's pure λ-calculus, which was designed as a pure calculus of functions in which other data types such as natural numbers can be encoded.

Action prefixes π define the communication behavior of processes. We have

$$\begin{array}{lcll} \pi & ::= & x(y) & \text{receive } y \text{ along } x \\ & \mid & \overline{x}\langle y\rangle & \text{send } y \text{ along } x \\ & \mid & \tau & \text{unobservable (internal) action} \end{array}$$

Process expressions P define the syntax of processes in the π-calculus. They rely on *sums* M, which represent a non-deterministic choice between processes waiting to perform an action (either input, output, or an internal action).

$$\begin{array}{lcll} P & ::= & M & \text{sum} \\ & \mid & \mathsf{0} & \text{termination} \\ & \mid & P_1 \mid P_2 & \text{composition} \\ & \mid & \mathsf{new}\ a\ P & \text{restriction} \\ & \mid & !P & \text{replication} \\ M & \mid & M_1 + M_2 & \text{choice} \\ & \mid & \pi.\,P & \text{guarded process} \end{array}$$

Milner now defines a *structural congruence* that identifies process expressions that are only distinguished by the limitations of syntax. For example, the process composition operator $P \mid Q$ should be commutative and associative so that a collection of concurrent processes can be written as $P_1 \mid \cdots \mid P_n$. Similarly, sums $M + N$ should be commutative and associative. $\mathsf{0}$ is the unit of composition so that a terminated process simply disappears.

Names require that we add the renaming of bound variables to our structural congruence. In particular, $\mathsf{new}\ a\ P$ binds a in P and $x(y).\,P$ binds y in P. Note that, conversely, $\overline{x}\langle y\rangle.\,P$ does *not* bind any variables: the name y is just sent along x. The order of consecutive bindings by $\mathsf{new}\ a$ may be changed, and we can extend or contract the scope of a $\mathsf{new}\ a$ binder across process composition as follows:

$$\mathsf{new}\ x\ (P \mid Q) \equiv P \mid (\mathsf{new}\ x\ Q)$$

[1] [*None of the material in this example has been proven correct at this time. Nor have we carefully surveyed the literature such as [BS92, Mil92].*]

provided x is not among the free names of P. This law of *scope extrusion* (read right to left) is important this it means a process can propagate a local names to its environment.

Finally, we have a rule of replication $!P \equiv P \mid !P$. Read from left to right it means a process $!P$ can replicate itself arbitrarily many times. From right to left the rule is of somewhat questionable value, since it would require recognizing structural equivalence of two active process expressions and then contracting them.

We will not formally model structural equivalence, because its necessary aspects will be captured by properties of the linear context Δ that contains active process expressions. Instead of repeating Milner's formal definition of the reaction rules, we explain them through their encoding in linear logic. The idea is the state of a process is represented by two proposition $\mathsf{proc}(P)$ for a process P and $\mathsf{choice}(M)$ for a sum M. A linear context

$$\Delta = \mathsf{proc}(P_1), \ldots, \mathsf{proc}(P_n), \mathsf{choice}(M_1), \ldots, \mathsf{choice}(M_m)$$

represents a state where processes P_i are executing concurrently and choices M_j are waiting to be made. Furthermore an unrestricted context

$$\Gamma = \mathsf{proc}(Q_1), \ldots, \mathsf{proc}(Q_p)$$

represents processes Q_k that may replicate themselves an arbitrary number of times. Informally, computation is modelled *bottom-up* in the sequent calculus, so that

$$\begin{array}{c} \Gamma_\pi, \Gamma_1; \Delta_1 \Longrightarrow C \\ \vdots \\ \Gamma_\pi, \Gamma_0; \Delta_0 \Longrightarrow C \end{array}$$

if we can transition from state Δ_0 with replicating processes Γ_0 to a state Δ_1 with replicating processes Γ_1. Here, C is arbitrary (in some sense, computation never stops) and Γ_π are the rules describing the legal reactions of the π-calculus as given below.

Process Composition $(P \mid Q)$**.** This just corresponds to a *fork* operation that generates two concurrently operating processes P and Q.

$$\mathsf{fork} : \mathsf{proc}(P \mid Q) \multimap \mathsf{proc}(P) \otimes \mathsf{proc}(Q)$$

Termination $\mathsf{0}$**.** This just corresponds to an *exit* operation, elimination the process.

$$\mathsf{exit} : \mathsf{proc}(\mathsf{0}) \multimap \mathbf{1}$$

Restriction $\mathsf{new}\ a\ P(a)$**.** The notation $P(a)$ represents a process P with some arbitrary number of occurrences of the bound variable a. We then write $P(x)$ for the result of substituting x for *all* occurrences of a in P. The new operation simply creates a new name, x, substitutes this for a in $P(a)$, and continues with

$P(x)$. The freshness condition on x can be enforced easily by the corresponding condition on the left rule for the existential quantifier $\exists$L in the sequent calculus.

$$\mathsf{gen} : \mathsf{proc}(\mathsf{new}\ a\ P(a)) \multimap \exists x.\ \mathsf{proc}(P(x))$$

While this is not completely formal at present, once we introduce the concept of *higher-order abstract syntax* in Section **??** we see that it can easily be modeled in linear logic.

Replication $!P$**.** This just moves the process into the unrestricted context so that as many copies of P can be generated by the use of the copy rule as needed.

$$\mathsf{promote} : \mathsf{proc}(!P) \multimap !\mathsf{proc}(P)$$

Coincidentally, this is achieved in linear logic with the "*of course*" modality that is also written as "!".

Sum M**.** A process expression that is a sum goes into a state where it can perform an action, either silent (τ) or by a reaction between input and output processes.

$$\mathsf{suspend} : \mathsf{proc}(M) \multimap \mathsf{choice}(M)$$

It is now very tempting to define choice simply as internal choice. That is,

$$?\mathsf{choose} : \mathsf{choice}(M_1 + M_2) \multimap \mathsf{choice}(M_1) \& \mathsf{choice}(M_2)$$

However, this does not correspond to semantics of the π-calculus. Instead, $M_1 + \cdots + M_n$ can perform an action if

1. either one of the M_i can perform a silent action τ in which case all alternatives M_j for $j \neq i$ are discarded,

2. or two guarded actions $x(y).\ P(y)$ and $\overline{x}\langle z\rangle.\ Q$ react, leaving processes $P(z)$ and Q while discarding all other alternatives.

We model this behavior with two auxiliary predicates $\mathsf{react}(M, N, P, Q)$ which is true if sums M and N can react, leaving processes P and Q, and $\mathsf{silent}(M, P)$ which is true if M can make a silent transition to P. These are invoked non-deterministically as follows:

$$\begin{array}{rcl} \mathsf{external} & : & \mathsf{choice}(M) \otimes \mathsf{choice}(N) \otimes !\mathsf{react}(M, N, P, Q) \multimap \mathsf{proc}(P) \otimes \mathsf{proc}(Q) \\ \mathsf{internal} & : & \mathsf{choice}(M) \otimes !\mathsf{silent}(M, P) \multimap \mathsf{proc}(P) \end{array}$$

Note the use of "!" before the react and silent propositions which indicates that proofs of these propositions do not refer to the current process state.

Reaction. Basic reaction is synchronous communication along a channel x. This is augmented by a rule to choose between alternatives.

$$\begin{array}{rcl} \mathsf{synch} & : & \mathsf{react}(x(y).\ P(y), \overline{x}\langle z\rangle.\ Q, P(z), Q) \\ \mathsf{choose}_2 & : & \mathsf{react}(M, N, P, Q) \multimap (\mathsf{react}(M + M_0, N, P, Q) \\ & & \qquad \& \mathsf{react}(M_0 + M, N, P, Q) \\ & & \qquad \& \mathsf{react}(M, N + N_0, P, Q) \\ & & \qquad \& \mathsf{react}(M, N_0 + M, P, Q)) \end{array}$$

Note that the synchronization rule synch again employs our notation for substitution.

Silent Action. A basic silent action simply discards the guard τ. This is augmented by a rule to choose between alternatives.

$$\begin{array}{rcl} \mathsf{tau} & : & \mathsf{silent}(\tau.\ P, P) \\ \mathsf{choose}_1 & : & \mathsf{silent}(M, P) \multimap (\mathsf{silent}(M + M_0, P) \& \mathsf{silent}(M_0 + M, P)) \end{array}$$

That's it! To model Milner's notion of structural equivalence faithfully we would need at least one other rule

$$?\mathsf{collect} : !\mathsf{proc}(P) \otimes \mathsf{proc}(P) \multimap !\mathsf{proc}(P)$$

but this is of questionable merit and rather an artefact of overloading the notion of structural congruence with too many tasks.

As a related example, we consider the asynchronous π-calculus without choice [HT91]. In this calculus we make two simplifications when compared to the (synchronous) π-calculus: (1) the sender of a message can proceed immediately without waiting for the receiver, and (2) there is no input choice construct. We therefore eliminate the syntactic category of sums. In addition, a message becomes like an independent process that can react with a guarded process waiting for input. Therefore we add a process expression $\overline{x}\langle y\rangle$ and code $\overline{x}\langle y\rangle.\ P$ as $\overline{x}\langle y\rangle \mid P$.

$$\begin{array}{rcll} P & ::= & \overline{x}\langle y\rangle & \text{message} \\ & | & x(y).\ P & \text{input action} \\ & | & \tau.\ P & \text{silent action} \\ & | & \mathsf{0} & \text{termination} \\ & | & P_1 \mid P_2 & \text{composition} \\ & | & \mathsf{new}\ a\ P & \text{restriction} \\ & | & !P & \text{replication} \end{array}$$

The encoding of the asynchronous π-calculus retains the rules fork, exit, gen and promote remain exactly as before. We replace the choice predicate entirely with two simple new rules: one rule for reaction (which is now essentially an input) and one rule for silent action.

$$\begin{array}{rcl}
\mathsf{fork} & : & \mathsf{proc}(P \mid Q) \multimap \mathsf{proc}(P) \otimes \mathsf{proc}(Q) \\
\mathsf{exit} & : & \mathsf{proc}(0) \multimap \mathbf{1} \\
\mathsf{gen} & : & \mathsf{proc}(\mathsf{new}\ a\ P(a)) \multimap \exists x.\ \mathsf{proc}(P(x)) \\
\mathsf{promote} & : & \mathsf{proc}(!P) \multimap !\mathsf{proc}(P) \\[4pt]
\mathsf{input} & : & \mathsf{proc}(\overline{x}\langle z\rangle) \otimes \mathsf{proc}(x(y).\ P(y)) \multimap \mathsf{proc}(P(z)) \\
\mathsf{silent} & : & \mathsf{proc}(\tau.\ P) \multimap \mathsf{proc}(P)
\end{array}$$

A variant of this encoding replaces input and silent by the following:

$$\begin{array}{rcl}
\mathsf{input}' & : & \mathsf{proc}(x(y).\ P(y)) \multimap (\forall y.\ \mathsf{proc}(\overline{x}\langle y\rangle) \multimap \mathsf{proc}(P(y))) \\
\mathsf{silent}' & : & \mathsf{proc}(\tau.\ P) \multimap (\mathbf{1} \multimap \mathsf{proc}(P))
\end{array}$$

In this representation, actions are not explitly part of the encoding. In particular, all linear implications have exactly one antecedent. Instead, we use the left rule for linear implication $\multimap$L in the sequent calculus to carry out the necessary steps. For more on this style of encoding, see Exercise 3.6.

One of the generalization that are generally needed for non-trivial examples is the *polyadic π-calculus* where an arbitrary number of names can be passed at once. The details of the encoding for the polyadic π-calulcus depend on details in the language of terms that we purposely glossed over in the presentation above. Many other variations have also been considered, such as the asynchronous π-calulcus with choice [ACS98] (see Exercise 3.7).

3.7 Exercises

Exercise 3.1 Consider if $\otimes$ and $\&$ can be distributed over $\oplus$ or *vice versa*. There are four different possible equivalences based on eight possible entailments. Give sequent derivations for the entailments that hold.

Exercise 3.2 Prove that the rule

$$\frac{(\Gamma, A\&B, A, B); \Delta \Longrightarrow C}{(\Gamma, A\&B); \Delta \Longrightarrow C}\,\&\mathrm{L}!$$

is admissible in the linear sequent calculus. Further prove that the rule

$$\frac{(\Gamma, A \otimes B, A, B); \Delta \Longrightarrow C}{(\Gamma, A \otimes B); \Delta \Longrightarrow C}\,\otimes\mathrm{L}!$$

is *not* admissible.

Determine which other connectives and constants have similar or analogous admissible rules directly on resource factories and which ones do not. You do not need to formally prove admissibility or unsoundness of your proposed rules.

Exercise 3.3 In the proof of admissibility of cut (Theorem 3.8) show the cases where

1. $\mathcal{D}$ ends in $\multimap$R and $\mathcal{E}$ ends in $\multimap$L and we have a principal cut.
2. $\mathcal{D}$ is arbitrary and $\mathcal{E}$ ends in $\multimap$L and we have a a right commutative cut.
3. $\mathcal{D}$ ends in !R and $\mathcal{E}$ and in !L and we have a principal cut.

Exercise 3.4 Reconsider the connective $A \multimapboth B$ from Exercise 2.9 which is true if A linearly implies B and vice versa.

- Give sequent calculus rules corresponding to your introduction and elimination rules.
- Show the new cases in the proof of soundness of the sequent calculus (Theorem 3.1).
- Show the new cases in the proof of completeness of the sequent calculus (Theorem 3.2).
- Show the new cases for principal cuts in the proof of admissibility of cut (Theorem 3.8).

Exercise 3.5 An extension of the notion of Petri net includes *inhibitor arcs* as inputs to a transition. An inhibitor arc lets a transition fire only if the place it is connected to does not contain any tokens. Show how to extend or modify the encoding of Petri nets from Section 3.2 so that it also models inhibitor arcs.

Exercise 3.6 The second representation of the asynchronous π-calculus used the clauses

$$\begin{array}{lcl} \mathsf{input}' & : & \mathsf{proc}(x(y).\,P(y)) \multimap (\forall y.\,\mathsf{proc}(\overline{x}\langle y\rangle) \multimap \mathsf{proc}(P(y))) \\ \mathsf{silent}' & : & \mathsf{proc}(\tau.\,P) \multimap (\mathbf{1} \multimap \mathsf{proc}(P)) \end{array}$$

to represent actions. Instead of *representing* the π-calculus by a single predicate proc and a number of unrestricted propositions that axiomatize transitions, this suggests a direct *embedding* of the asynchronous π-calculus in linear logic. Here is a start for this kind of embedding, denoted by $(\ )^*$.

$$\begin{array}{rcl} (0)^* & = & \mathbf{1} \\ (P \mid Q)^* & = & P^* \otimes Q^* \\ (!P)^* & = & !P^* \end{array}$$

Complete this embedding by giving translations of the remaining constructs of the asynchronous π-calculus as propositions in linear logic. Carefully state the adequacy theorem for your representation. [Extra Credit: Prove the adequacy theorem.]

Exercise 3.7 Extend one of the encodings of the asynchronous π-calculus to allow input choice as present in the synchronous π-calculus.

Chapter 4

Proof Search

Linear logic as introduced by Girard and presented in the previous chapter is a rich system for the formalization of reasoning involving state. It conservatively extends intuitionistic logic and can therefore also serve as the logical basis for general constructive mathematics. Searching for proofs in such an expressive logic is difficult, and one should not expect silver bullets.

Depending on the problem, proof search in linear logic can have a variety of applications. In the domain of planning problems (see Section 2.4) searching for a proof means searching for a plan. In the domain of concurrent computation (see Petri nets in Section 3.2 or the π-calculus in Section 3.6) searching for a proof means searching for possible computations. In the domain of logic programming (which we investigate in detail in Chapter 5), searching for a proof according to a fixed strategy is the basic paradigm of computation. In the domain of functional programming and type theory (which we investigate in Chapter 6), searching for a proof means searching for a program satisfying a given specification.

Each application imposes different requirements on proof search, but there are underlying basic techniques which recur frequently. In this chapter we take a look at some basic techniques, to be exploited in subsequent chapters.

4.1 Bottom-Up Proof Search and Inversion

The literature is not in agreement on the terminology, but we refer to the process of creating a derivation from the desired judgment on upward as *bottom-up* proof search. A snap-shot of a bottom-up search is a partial derivation, with undecided judgments at the top. Our goal is to derive all remaining judgments, thereby completing a proof.

We proceed by selecting a judgment which remains to be derived and an inference rule with which it might be inferred. We also may need to determine exactly how the conclusion of the rule matches the judgment. For example, in the $\otimes$R rule we need to decide how to split the linear hypotheses between the two premises. After these choices have been made, we reduce the goal of deriving the judgment to a number of subgoals, one for each premise of the selected rule. If there are no premises, the subgoal is solved. If there are no subgoals left, we have derived the original judgment.

One important observation about bottom-up proof search is that some rules are *invertible*, that is, the premises are derivable whenever the conclusion is derivable. The usual direction states that the conclusion is evident whenver the premises are. Invertible rules can safely be applied whenever possible without losing completeness, although some care must be taken to retain a terminating procedure in the presence of unrestricted hypotheses. We also separate *weakly invertible* rules, which only apply when there are no linear hypotheses (besides possibly the principal proposition of the inference rule). For example, we cannot apply the rule $\mathbf{1}$R whenever the judgment is $\Gamma; \Delta \vdash \mathbf{1}$, although it is safe to do so when there are no linear hypotheses. Similarly, we cannot use the initial sequent rule to infer $\Gamma; \Delta, A \Longrightarrow A$ unless $\Delta = \cdot$. Strongly invertible rules apply regardless of any other hypotheses.

Theorem 4.1 (Inversion Lemmas) *The following table lists invertible, weakly invertible, and non-invertible rule in intuitionistic linear logic.*

Strongly Invertible	*Weakly Invertible*	*Not Invertible*
$\multimap$R		$\multimap$L
$\otimes$L, $\mathbf{1}$L	$\mathbf{1}$R	$\otimes$R
&R, $\top$R		&L_1, &L_2
$\oplus$L, $\mathbf{0}$L		$\oplus\mathrm{R}_1$, $\oplus\mathrm{R}_2$
$\forall$R, $\exists$L		$\forall$L, $\exists$R
$\supset$R, !L	!R	$\supset$L

We exclude the init *and* copy *rules, since they are neither proper left nor proper right rules.*

Proof: For invertible rules we prove that each premise follows from the conclusion. For non-invertible rules we give a counterexample. The two sample case below are representative: for invertible rules we apply admissibility of cut, for non-invertible rules we consider a sequent with the same proposition on the left and right.

Case: $\multimap$R is invertible. We have to show that $\Gamma; (\Delta, A) \Longrightarrow B$ is derivable whenver $\Gamma; \Delta \Longrightarrow A \multimap B$ is derivable, so we assume $\Gamma; \Delta \Longrightarrow A \multimap B$. We also have $\Gamma; A, A \multimap B \Longrightarrow B$, which follows by one $\multimap$L rule from two initial sequents. From the admissibility of cut (Theorem 3.8) we then obtain directly $\Gamma; (\Delta, A) \Longrightarrow B$.

Case: $\multimap$L is not invertible. Consider $\cdot; A \multimap B \Longrightarrow A \multimap B$ for parameters A and B. There is only one way to use $\multimap$L to infer this, which leads to $\cdot; \cdot \Longrightarrow A$ and $\cdot; B \Longrightarrow A \multimap B$, neither of which is derivable. Therefore $\multimap$L is not invertible in general.

□

As a final, general property for bottom-up proof search we show that we can restrict ourselves to initial sequents of the form $\Gamma; P \Longrightarrow P$, where P is an atomic proposition. We write $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$ for the restricted judgment whose rules are as for $\Gamma; \Delta \Longrightarrow A$, except that initial sequents are restricted to atomic propositions. Obviously, if $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$ then $\Gamma; \Delta \Longrightarrow A$.

Theorem 4.2 (Completeness of Atomic Initial Sequents) *If* $\Gamma; \Delta \Longrightarrow A$ *then* $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$.

Proof: By induction on the the structure of $\mathcal{D} :: (\Gamma; \Delta \Longrightarrow A)$. In each case except initial sequents, we appeal directly to the induction hypothesis and infer $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$ from the results. For initial sequents, we use an auxiliary induction on the structure of the proposition A. We show only one case—the others are similar in that they follow the local expansions, translated from natural deduction to the setting of the sequent calculus. If local completeness did not hold for a connective, then atomic initial sequents would be incomplete as well.

Case: $A = A_1 \otimes A_2$. Then we construct

$$\cfrac{\cfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma; A_1 \stackrel{-}{\Longrightarrow} A_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Gamma; A_2 \stackrel{-}{\Longrightarrow} A_2\end{array}}{\Gamma; A_1, A_2 \stackrel{-}{\Longrightarrow} A_1 \otimes A_2}\,\otimes\mathrm{R}}{\Gamma; A_1 \otimes A_2 \stackrel{-}{\Longrightarrow} A_1 \otimes A_2}\,\otimes\mathrm{L}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ exist by induction hypothesis on A_1 and A_2.

□

The theorems in this section lead to a search procedure with the following general outline:

1. Pick a goal sequent to solve.
2. Decide to apply a right rule to the consequent or a left rule to a hypothesis.
3. Determine the remaining parameters (either how to split the hypotheses, or on the terms which may be required).
4. Apply the rule in the backward direction, reducing the goal to possibly several subgoals.

A lot of choices remain in this procedure. They can be classified according to the type of choice which must be made. This classification will guide us in the remainder of this chapter, as we discuss how to reduce the inherent non-determinism in the procedure above.

- Conjunctive choices. We know all subgoals have to be solved, but the order in which we attempt to solve them is not determined. In the simplest case, this is a form of *don't-care non-determinism*, since all subgoals have to be solved. In practice, it is not that simple since subgoals may interact once other choices have been made more deterministic. Success is a special case of conjunctive choice with no conjuncts.

- Disjunctive choices. We don't know which left or right rule to apply. Invertible rules are always safe, but once they all have been applied, many possibilities may remain. This is a form of *don't-know non-determinism*, since a sequence of correct guesses will lead to a derivation if there is one. In practice, this may be solved via backtracking, for example. Failure is a special case of a disjunctive choice with zero alternatives.

- Resource choices. We do not know how to divide our resources in the multiplicative rules. This is a special case of don't-know non-determinism which can be solved by different techniques collectively referred to as "resource managment". Resource management interacts tightly with other disjunctive and conjunctive choices.

- Universal choices. In the $\forall$R and $\exists$L rules we have to choose a new parameter. Fortunately, this is a trivial choice, since *any* new parameter will work, and its name is not important. Hence this is a form of don't-care non-determinism.

- Existential choices. In the $\exists$R and $\forall$L rules we have to choose a term t to substitute for the bound variable. Since there are potentially infinitely many terms (depending on the domain of quantification), this is a form of don't-know non-determinism. In practice, this is solved by *unification*, discussed in Section 4.3.

4.2 Focusing

Focusing combines two basic phases in order to reduce non-determinism in proof search while remaining sound and complete.

1. (Inversion) Strongly invertible rules are applied eagerly. The order of these rule applications does not matter, so this is an instance of don't-care non-determinism.

2. (Focusing) After some steps we arrive at a sequent where all applicable rules with the exception of copy or init are non-invertible. Now we *focus*, either on a particular hypothesis or on the conclusion and apply a sequence of non-invertible rules until we have exposed an invertible principle connective. At this point in the proof search we return to the inversion phase.

We refer to this strategy as *focused proof search*. The idea and method are due to Andreoli [And92]; it is closely related to logic programming and the notion of *uniform proof* [MNPS91] as we will see in Chapter 5.

Just like the sequent calculus followed inevitably from natural deduction, focused proof search seems to follow inevitably from the sequent calculus. It is remarkably robust in that in our experience, any logic that admits a clean sequent calculus also admits a similarly clean focusing calculus. This is true even for logics such as classical logic for which good natural deduction systems that arise from judgmental considerations are elusive.

While the basic intuition is simple, giving an unambiguous specification of focusing is a non-trivial task. Both the proper representation of the don't-care non-determinism and the notion of focus proposition for phase (2) require some experience and (eventually) lengthy correctness proofs.

In order to aid the description of the rules, we define some classes of propositions. We say A is *right asynchronous* if the top-level connective of A has a strongly invertible right rule. Similarly, A is *left asynchronous* if the top-level connective of A has a strongly invertible left rule. The intuition is that of asynchronous communication, where a sending process can proceed immediately without waiting for receipt of its message. Dually, a proposition is *right* or *left synchronous* is its top-level connective has a non-invertible or only weakly invertible right or left rule, respectively.

Atomic	P
Right Asynchronous	$A_1 \multimap A_2, A_1 \& A_2, \top, A_1 \supset A_2, \forall x.\ A$
Left Asynchronous	$A_1 \otimes A_2, \mathbf{1}, A_1 \oplus A_2, \mathbf{0}, !A, \exists x.\ A$
Right Synchronous	$A_1 \otimes A_2, \mathbf{1}, A_1 \oplus A_2, \mathbf{0}, !A, \exists x.\ A$
Left Synchronous	$A_1 \multimap A_2, A_1 \& A_2, \top, A_1 \supset A_2, \forall x.\ A$

Note that the left asynchronous and right synchronous propositions are identical, as are the right asynchronous and left synchronous. We therefore really need only two classes of propositions, but this tends to be very confusing.

Inversion. The first phase of proof search decomposes all asynchronous connectives. This means there is a lot of don't-care non-determinism, since the order in which the rules are applied is irrelevant. We build this into our system by *fixing* a particular order in which the asynchronous connectives are decomposed: first the succedent, then the antecedents from right to left. This means we need a new form of hypothetical judgment, an *ordered hypothetical judgment*, since otherwise we cannot prescribe a fixed order. We thoroughly treat this judgment form in Chapter **??**. We write

$$\Gamma; \Delta; \Omega \Longrightarrow A \Uparrow$$

where

Γ are unrestricted hypotheses (which may be arbitrary),
Δ are linear hypotheses that *may not be left asynchronous*,
Ω are ordered hypotheses (which may be arbitrary),
A is the goal (which may be arbitrary).

The first set of rules treats all right asynchronous connectives.

$$\frac{\Gamma; \Delta; \Omega, A \Longrightarrow B \Uparrow}{\Gamma; \Delta; \Omega \Longrightarrow A \multimap B \Uparrow} \multimap R \qquad \frac{\Gamma; \Delta; \Omega \Longrightarrow A \Uparrow \qquad \Gamma; \Delta; \Omega \Longrightarrow B \Uparrow}{\Gamma; \Delta; \Omega \Longrightarrow A \& B \Uparrow} \& R$$

$$\frac{}{\Gamma; \Delta; \Omega \Longrightarrow \top \Uparrow} \top R \qquad \frac{\Gamma, A; \Delta; \Omega \Longrightarrow B \Uparrow}{\Gamma; \Delta; \Omega \Longrightarrow A \supset B \Uparrow} \supset R$$

$$\frac{\Gamma; \Delta; \Omega \Longrightarrow [a/x]A \Uparrow}{\Gamma; \Delta; \Omega \Longrightarrow \forall x.\, A \Uparrow} \forall R^a$$

Once the goal proposition is no longer asynchronous, we proceed with the hypotheses in Ω, decomposing all left asynchronous propositions in them. We write

$$\Gamma; \Delta; \Omega \Uparrow \Longrightarrow C$$

where

Γ are unrestricted hypotheses (arbitrary)
Δ are linear hypotheses (not left asynchronous)
Ω are ordered hypotheses (arbitrary)
C is the goal (not right asynchronous)

First, we have the rule to transiton to this judgment.

$$\frac{\Gamma; \Delta; \Omega \Uparrow \Longrightarrow C, \quad C \text{ not right asynchronous}}{\Gamma; \Delta; \Omega \Longrightarrow C \Uparrow} \Uparrow R$$

Next we have rules to decompose the left asynchronous proposition in Ω *in order*, that is, at the right end of the ordered hypotheses.

$$\frac{\Gamma; \Delta; \Omega, A, B \Uparrow \Longrightarrow C}{\Gamma; \Delta; \Omega, A \otimes B \Uparrow \Longrightarrow C} \otimes L \qquad \frac{\Gamma; \Delta; \Omega \Uparrow \Longrightarrow C}{\Gamma; \Delta; \Omega, \mathbf{1} \Uparrow \Longrightarrow C} \mathbf{1}L$$

$$\frac{\Gamma; \Delta; \Omega, A \Uparrow \Longrightarrow C \qquad \Gamma; \Delta; \Omega, B \Uparrow \Longrightarrow C}{\Gamma; \Delta; \Omega, A \oplus B \Uparrow \Longrightarrow C} \oplus L \qquad \frac{}{\Gamma; \Delta; \Omega, \mathbf{0} \Uparrow \Longrightarrow C} \mathbf{0}L$$

$$\frac{\Gamma, A; \Delta; \Omega \Uparrow \Longrightarrow C}{\Gamma; \Delta; \Omega, !A \Uparrow \Longrightarrow C} !L \qquad \frac{\Gamma; \Delta; \Omega, [a/x]A \Uparrow \Longrightarrow C}{\Gamma; \Delta; \Omega, \exists x.\, A \Uparrow \Longrightarrow C} \exists L^a$$

When we encounter a proposition that is not left asynchronous, we move it into Δ so that we can eventually eliminate all propositions from Ω.

$$\frac{\Gamma; \Delta, A; \Omega \Uparrow \Longrightarrow C, \quad A \text{ not left asynchrounous}}{\Gamma; \Delta; \Omega, A \Uparrow \Longrightarrow C} \Uparrow L$$

Decision. When we start from $\Gamma; \cdot; \Omega \Longrightarrow A \Uparrow$, searching backwards, we can always reach a situation of the form $\Gamma'; \Delta'; \cdot \Uparrow \Longrightarrow C$ for each leaf where C is not right asynchronous and Δ' contains no propositions that are left asynchronous. At this point we need to decide which proposition to focus on. Note that we always focus on a single proposition. First the case we focus on the right. For this we need a new judgment

$$\Gamma; \Delta; \cdot \Longrightarrow A \Downarrow$$

where

Γ are unrestricted hypotheses (arbitrary),
Δ are linear hypotheses (not left asynchronous),
A is the focus proposition (arbitrary).

If we focus on the left, we need an analogous judgment.

$$\Gamma; \Delta; A \Downarrow \Longrightarrow C$$

where

Γ are unrestricted hypotheses (arbitary),
Δ are linear hypotheses (not left asynchronous),
A is the focus proposition (arbitrary),
C is the succedent (not right asynchronous)

The decision is between the following rules that transition into these two jugments.

$$\frac{\Gamma; \Delta; \cdot \Longrightarrow C \Downarrow, \quad C \text{ not atomic}}{\Gamma; \Delta; \cdot \Uparrow \Longrightarrow C}\,\mathsf{decideR}$$

$$\frac{\Gamma; \Delta; A \Downarrow \Longrightarrow C}{\Gamma; \Delta, A; \cdot \Uparrow \Longrightarrow C}\,\mathsf{decideL} \qquad \frac{\Gamma, A; \Delta; A \Downarrow \Longrightarrow C}{\Gamma, A; \Delta; \cdot \Uparrow \Longrightarrow C}\,\mathsf{decideL!}$$

Note that $\mathsf{decideL!}$ is justified by the copy rule.

Focusing. Once we have decided which proposition to focus on, we apply a succession of non-invertible rules.

$$\frac{\Gamma; \Delta_1; \cdot \Longrightarrow A_1 \Downarrow \qquad \Gamma; \Delta_2; \cdot \Longrightarrow A_2 \Downarrow}{\Gamma; \Delta_1, \Delta_2; \cdot \Longrightarrow A_1 \otimes A_2 \Downarrow}\,\otimes\mathrm{R} \qquad \frac{}{\Gamma; \cdot; \cdot \Longrightarrow \mathbf{1} \Downarrow}\,\mathbf{1}\mathrm{R}$$

$$\frac{\Gamma; \Delta; \cdot \Longrightarrow A \Downarrow}{\Gamma; \Delta; \cdot \Longrightarrow A \oplus B \Downarrow}\,\oplus\mathrm{R}_1 \qquad \frac{\Gamma; \Delta; \cdot \Longrightarrow B \Downarrow}{\Gamma; \Delta; \cdot \Longrightarrow A \oplus B \Downarrow}\,\oplus\mathrm{R}_2$$

$$\text{no right rule for } \mathbf{0} \qquad \frac{\Gamma; \Delta; \cdot \Longrightarrow [t/x]A \Downarrow}{\Gamma; \Delta; \cdot \Longrightarrow \exists x.\, A \Downarrow}\,\exists\mathrm{R}^a$$

$$\frac{\Gamma; \cdot; \cdot \Longrightarrow A \Uparrow}{\Gamma; \cdot; \cdot \Longrightarrow {!A} \Downarrow}\,!\mathrm{R}$$

The last rule is a somewhat special case: because !R is weakly right invertible, we immediately transition back to break down the right asynronous connectives in A. In the other weakly right invertible rule, $\mathbf{1}$R, we conclude the proof so no special provision is necessary.

The corresponding left rules are as follows:

$$\frac{\Gamma; \Delta_2; B \Downarrow \Longrightarrow C \qquad \Gamma; \Delta_1; \cdot \Longrightarrow A \Downarrow}{\Gamma; \Delta_1, \Delta_2; A \multimap B \Downarrow \Longrightarrow C} \multimap \mathrm{L}$$

$$\frac{\Gamma; \Delta; A \Downarrow \Longrightarrow C}{\Gamma; \Delta; A \& B \Downarrow \Longrightarrow C} \&\mathrm{L}_1 \qquad \frac{\Gamma; \Delta; B \Downarrow \Longrightarrow C}{\Gamma; \Delta; A \& B \Downarrow \Longrightarrow C} \&\mathrm{L}_2$$

$$\text{no left rule for } \top \qquad \frac{\Gamma; \Delta; B \Downarrow \Longrightarrow C \qquad \Gamma; \cdot; \cdot \Longrightarrow A \Uparrow}{\Gamma; \Delta; A \supset B \Downarrow \Longrightarrow C} \supset\mathrm{L}$$

$$\frac{\Gamma; \Delta; [t/x]A \Downarrow \Longrightarrow C}{\Gamma; \Delta; \forall x.\, A \Downarrow \Longrightarrow C} \forall\mathrm{L}$$

Eventually we must break down the focus proposition to the point where it is no longer synchronous. If it is atomic, we either succeed or fail in our overall proof attempt. In other cases we switch back to the inversion judgment.

$$\frac{}{\Gamma; \cdot; P \Downarrow \Longrightarrow P} \mathsf{init}$$

$$\frac{\Gamma; \Delta; A \Uparrow \Longrightarrow C \quad A \text{ not atomic and not left synchronous}}{\Gamma; \Delta; A \Downarrow \Longrightarrow C} \Downarrow\mathrm{L}$$

$$\frac{\Gamma; \Delta; \cdot \Longrightarrow A \Uparrow}{\Gamma; \Delta \Longrightarrow A \Downarrow} \Downarrow\mathrm{R}$$

The soundness of these rules is relatively easy to establish, since, in the end, the system just represents a restriction on the application of the usual left, right, initial and copy rules. Completeness of the corresponding system for classical linear logic has been proven by Andreoli [And92] and is lengthy. The completeness of the rules above has not yet been considered, but Andreoli's techniques would seem to apply fairly directly.[1]

In order to state soundness formally, we use convention that Δ, Ω joins the contexts Δ and Ω, ignoring the order of the hypotheses in Ω.

Theorem 4.3 (Soundness of Focusing)

1. *If* $\Gamma; \Delta; \Omega \Longrightarrow A \Uparrow$ *then* $\Gamma; (\Delta, \Omega) \Longrightarrow A$.
2. *If* $\Gamma; \Delta; \Omega \Uparrow \Longrightarrow C$ *then* $\Gamma; (\Delta, \Omega) \Longrightarrow C$.

[1] [*This might make an interesting class project.*]

3. *If* $\Gamma; \Delta; \cdot \Longrightarrow A \Downarrow$ *then* $\Gamma; \Delta \Longrightarrow A$.

4. *If* $\Gamma; \Delta; A \Downarrow \Longrightarrow C$ *then* $\Gamma; (\Delta, A) \Longrightarrow C$

Proof: By straightforward simultaneous induction on the structure of the given deductions. □

Focusing eliminates a lot of non-determinism regarding the choices among possible inference rules. We summarize again the essential improvements over the plain sequent calculus.

Inversion. Invertible rules are applied eagerly without considering alternatives. This does not jeopardize completeness precisely because these rules are (strongly) invertible. The introduction of ordered hypotheses Ω allows eliminates the don't-care non-determinism arises from the choice of invertible rule by fixing an order. It is an important property of the focusing system that in fact the order is irrelevant in the end, i.e., exchange between ordered hypotheses is admissible for the focusing calculus.

Focusing. Once a non-invertible rule has been selected, a sequence of non-invertible rules is applied to the focus proposition. This may involve don't-know non-deterministic choices, such as the question which side of a disjunction may be proven. However, other choices (such as interleaving left rules between independent linear hypotheses) are eliminated from consideration.

Initial Sequents. When we focus on a hypothesis, we break down the principal connective of the focus proposition until is has been reduced to either invertible or atomic. In case it is atomic, it must match an atomic goal—otherwise we fail. This improvement will be particularly important under the logic programming interpretation of linear logic.

As listed at the end of the last section, this still leaves a significant amount of non-determinism which needs to be controlled during proof search. Instead of piling these improvements on top of each other, we consider each one separately. In a realistic application, such as general theorem proving, logic programming, model checking, many of these high-level optimizations may need to be applied simultaneously and their interaction considered.

4.3 Unification

We begin with a discussion of *unification*, a technique for eliminating existential non-determinism. When proving a proposition of the form $\exists x.\ A$ by its right rule in the sequent calculus, we must supply a term t and then prove $[t/x]A$. The domain of quantification may include infinitely many terms (such as the natural numbers), so this choice cannot be resolved simply by trying all possible terms t. Similarly, when we use a hypothesis of the form $\forall x.\ A$ we must supply a term t to substitute for x.

Fortunately, there is a better technique which is sound and complete for syntactic equality between terms. The basic idea is quite simple: we postpone the choice of t and instead substitute a new *existential variable* (often called *meta-variable* or *logic variable*) X for x and continue with the bottom-up construction of a derivation. When we reach initial sequents we check if there is a substitution for the existential variables such that the hypothesis matches the conclusion. If so, we apply this instantiation globally to the partial derivation and continue to search for proofs of other subgoals. Finding an instantiation for existential variables under which two propositions or terms match is called *unification*. It is decidable if a unifying substitution or *unifier* exists, and if so, we can effectively compute it in linear time. Moreover, we can do so with a minimal commitment and we do not need to choose between various possible unifiers.

Because of its central importance, unification has been thoroughly investigated. Herbrand [Her30] is given credit for the first description of a unification algorithm in a footnote of his thesis, but it was not until 1965 that it was introduced into automated deduction through the seminal work by Alan Robinson [Rob65, Rob71]. The first algorithms were exponential, and later almost linear [Hue76, MM82] and linear algorithms [MM76, PW78] were discovered. In the practice of theorem proving, generally variants of Robinson's algorithm are still used, due to its low constant overhead on the kind of problems encountered in practice. For further discussion and a survey of unification, see [Kni89]. We describe a variant of Robinson's algorithm.

Before we describe the unification algorithm itself, we relate it to the problem of proof search. For this we use a general method of *residuation*. It should be kept in mind that this is mostly a foundational tool and not necessarily a direct path to an implementation. We enrich the judgment $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$ by a *residual proposition* F such that

1. if $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$ then $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A \setminus F$ and F is true, and

2. if $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A \setminus F$ and F is true then $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$.

Generally, we cannot prove such properties directly by induction, but we need to generalize them, exhibiting the close relationship between the derivations of the sequents and residual formulas F.

Residual formulas F are amenable to specialized procedures such as unification, since they are drawn from a simpler logic or deductive system than the general propositions A. In practice they are often solved *incrementally* rather than collected throughout a derivation and only solved at the end. This is important for the early detection of failures during proof search. Incremental solution of residual formulas is the topic of Exercise **??**.

What do we need in the residual propositions so that existential choices and equalities between atomic propositions can be expressed? The basic proposition is one of equality between atomic propositions, $P_1 \doteq P_2$. We also have conjunction $F_1 \wedge F_2$, since equalities may be collected from several subgoals, and $\top$ if there are no residual propositions to be proven. Finally, we need the existential quantifier $\exists x.\, F$ to express the scope of existential variables, and $\forall x.\, F$ to express the scope of parameters introduced in a derivation. We add equality between terms, since it is required to describe the unification algorithm itself. We refer to the logic with these connectives as *unification logic*, defined via a deductive system.

$$\textit{Formulas} \quad F \quad ::= \quad P_1 \doteq P_2 \mid t_1 \doteq t_2 \mid F_1 \wedge F_2 \mid \top \mid \exists x.\, F \mid \forall x.\, F$$

The main judgment "*F is valid*", written $\models F$, is defined by the following rules, which are consistent with, but more specialized than the rules for these connectives in intuitionistic natural deduction (see Exercise **??**).

$$\frac{}{\models P \doteq P}\doteq\mathrm{I} \qquad\qquad \frac{}{\models t \doteq t}\doteq\mathrm{I}'$$

$$\frac{\models F_1 \qquad \models F_2}{\models F_1 \wedge F_2}\wedge\mathrm{I} \qquad\qquad \frac{}{\models \top}\top\mathrm{I}$$

$$\frac{\models [t/x]F}{\models \exists x.\, F}\exists\mathrm{I} \qquad\qquad \frac{\models [a/x]F}{\models \forall x.\, F}\forall\mathrm{I}^a$$

The $\forall\mathrm{I}^a$ rule is subject to the usual proviso that a is a new parameter not occurring in $\forall x.\, F$. There are no elimination rules, since we do not need to consider hypotheses of the form $\models F$, which is the primary reason for the simplicity of theorem proving in the unification logic.

We enrich the sequent calculus with residual formulas from the unification logic, postponing all existential choices. Recall that in practice we merge residuation and solution in order to discover unprovable residual formulas as soon as possible. This merging of the phases is not represented in our system.

Hypotheses. Initial sequents residuate an equality between its principal propositions. Any solution to the equation will unify P' and P, which means that this will translate to a correct application of the initial sequent rule in the original system.

$$\frac{}{\Gamma; P' \stackrel{-}{\Longrightarrow} P \setminus P' \doteq P}\mathsf{init} \qquad\qquad \frac{(\Gamma, A); (\Delta, A) \stackrel{-}{\Longrightarrow} C \setminus F}{(\Gamma, A); \Delta \stackrel{-}{\Longrightarrow} C \setminus F}\mathsf{copy}$$

Propositional Connectives. We just give a few sample rules for the connectives which do not involve quantifiers, since all of them simply propagate or combine unification formulas, regardless whether they are additive, multiplicative, or exponential.

$$\frac{\Gamma; \Delta, A \stackrel{-}{\Longrightarrow} B \setminus F}{\Gamma; \Delta \stackrel{-}{\Longrightarrow} A \multimap B \setminus F} \multimap R \qquad \frac{\Gamma; \Delta_1 \stackrel{-}{\Longrightarrow} A \setminus F_1 \qquad \Gamma; \Delta_2, B \stackrel{-}{\Longrightarrow} C \setminus F_2}{\Gamma; \Delta_1, \Delta_2, A \multimap B \stackrel{-}{\Longrightarrow} C \setminus F_1 \wedge F_2} \multimap L$$

$$\frac{}{\Gamma; \cdot \stackrel{-}{\Longrightarrow} \mathbf{1} \setminus \top} \mathbf{1}R \qquad \frac{\Gamma; \Delta \stackrel{-}{\Longrightarrow} C \setminus F}{\Gamma; \Delta, \mathbf{1} \stackrel{-}{\Longrightarrow} C \setminus F} \mathbf{1}L$$

Quantifiers. These are the critical rules. Since we residuate the existential choices entirely, the $\exists$R and $\forall$L rules instantiate a quantifier by a new *parameter*, which is existentially quantified in the residual formula in both cases. Similarly, the $\forall$R and $\exists$L rule introduce a parameter which is universally quantified in the residual formula.

$$\frac{\Gamma; \Delta \stackrel{-}{\Longrightarrow} [a/x]A \setminus [a/x]F}{\Gamma; \Delta \stackrel{-}{\Longrightarrow} \forall x.\, A \setminus \forall x.\, F} \forall R^a \qquad \frac{\Gamma; \Delta, [a/x]A \stackrel{-}{\Longrightarrow} C \setminus [a/x]F}{\Gamma; \Delta, \forall x.\, A \stackrel{-}{\Longrightarrow} C \setminus \exists x.\, F} \forall L^a$$

$$\frac{\Gamma; \Delta \stackrel{-}{\Longrightarrow} [a/x]A \setminus [a/x]F}{\Gamma; \Delta \stackrel{-}{\Longrightarrow} \exists x.\, A \setminus \exists x.\, F} \exists R^a \qquad \frac{\Gamma; \Delta, [a/x]A \stackrel{-}{\Longrightarrow} C \setminus [a/x]F}{\Gamma; \Delta, \exists x.\, A \stackrel{-}{\Longrightarrow} C \setminus \forall x.\, A} \exists L^a$$

The soundness of residuating equalities and existential choices in this manner is straightforward.

Theorem 4.4 (Soundness of Equality Residuation) *If* $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A \setminus F$ *and* $\models F$ *then* $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$*.*

Proof: By induction on the structure of $\mathcal{R} :: (\Gamma; \Delta \stackrel{-}{\Longrightarrow} A \setminus F)$. We show the critical cases. Note how in the case of the $\exists$R rule the proof of $\models \exists x.\, F$ provides the essential witness term t.

Case: $\mathcal{R} = \dfrac{}{\Gamma; P' \stackrel{-}{\Longrightarrow} P \setminus P' \doteq P} \mathrm{I}.$

We know by assumption that $\models F$ which reads $\models P' \doteq P$. By inversion therefore $P' = P$ (since $\doteq$I is the only rule which applies to this judgment), and $\Gamma; P' \stackrel{-}{\Longrightarrow} P$ is a valid initial sequent.

Case: $\mathcal{R} = \dfrac{\begin{array}{c}\mathcal{R}_1\\ \Gamma; \Delta \stackrel{-}{\Longrightarrow} [a/x]A_1 \setminus [a/x]F_1\end{array}}{\Gamma; \Delta \stackrel{-}{\Longrightarrow} \exists x.\, A_1 \setminus \exists x.\, F_1} \exists R^a.$

By assumption, we have $\models \exists x.\, F_1$. By inversion, $\models [t/x]F_1$ for some t. By the proviso on the $\exists R^a$ rule, $\mathcal{R}_1$ is parametric in a, so we can substitute

t for a in this derivation an obtain $[t/a]\mathcal{R}_1 :: (\Gamma; \Delta \stackrel{-}{\Longrightarrow} [t/x]A_1 \setminus [t/x]F_1)$. Applying the induction hypothesis to $[t/a]\mathcal{R}_1$ yields a $\mathcal{D}_1$ and we construct

$$\frac{\begin{array}{c}\mathcal{D}_1\\ \Gamma; \Delta \stackrel{-}{\Longrightarrow} [t/x]A_1\end{array}}{\Gamma; \Delta \stackrel{-}{\Longrightarrow} \exists x.\ A_1}\ \exists\mathrm{R}$$

Case: $\mathcal{R} = \dfrac{\begin{array}{c}\mathcal{R}_1\\ \Gamma; \Delta \stackrel{-}{\Longrightarrow} [a/x]A_1 \setminus [a/x]F_1\end{array}}{\Gamma; \Delta \stackrel{-}{\Longrightarrow} \forall x.\ A_1 \setminus \forall x.\ F_1}\ \forall\mathrm{R}^a.$

By assumption, we have $\models \forall x.\ F_1$. By inversion, $\models [b/x]F_1$ for a new parameter b, and therefore also $\models [a/x]F_1$ by substitution. Hence we can apply the induction hypothesis to obtain a $\mathcal{D}_1$ and construct

$$\frac{\begin{array}{c}\mathcal{D}_1\\ \Gamma; \Delta \stackrel{-}{\Longrightarrow} [a/x]A_1\end{array}}{\Gamma; \Delta \stackrel{-}{\Longrightarrow} \forall x.\ A_1}\ \forall\mathrm{R}^a$$

□

The opposite direction is more difficult. The desired theorem:

If $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$ *then* $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A \setminus F$ *for some* F *with* $\models F$

cannot be proved directly by induction, since the premises of the two derivations are different in the $\exists$R and $\forall$L rules. However, one can be obtained from the other by substituting terms for parameters. Since this must be done simultaneously, we introduce a new notation.

$$\textit{Parameter Substitution} \quad \rho \quad ::= \quad \cdot \mid \rho, t/a$$

We assume all the parameters a substituted for by ρ are distinct to avoid ambiguity. We write $[\rho]A$, $[\rho]F$, and $[\rho]\Gamma$, for the result of applying the substitution ρ to a proposition, formula, or context, respectively.

Lemma 4.5 *If* $\Gamma; \Delta \stackrel{-}{\Longrightarrow} A$ *and* $[\rho]A' = A$, $[\rho]\Delta' = \Delta$, *and* $[\rho]\Gamma' = \Gamma$, *then* $\Gamma'; \Delta' \stackrel{-}{\Longrightarrow} A' \setminus F$ *for some* F *and* $\models [\rho]F$.

Proof: The proof proceeds by induction on the structure of $\mathcal{D} :: (\Gamma; \Delta \stackrel{-}{\Longrightarrow} A)$. We show only three cases, the second of which required the generalization of the induction hypothesis.

Case: $\mathcal{D} = \dfrac{}{\Gamma; P \overset{=}{\Longrightarrow} P}\, I$

and $[\rho]\Gamma' = \Gamma$, $[\rho]\Delta' = P$, and $[\rho]A' = P$. Therefore $\Delta' = P''$ with $[\rho]P'' = P$ and $A' = P'$ with $[\rho]P' = P$ and we construct

$$\frac{}{\Gamma'; P'' \overset{=}{\Longrightarrow} P' \setminus P'' \doteq P'}\, I \quad \text{and} \quad \frac{}{\models [\rho]P'' \doteq [\rho]P'} \doteq\mathrm{I}$$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma; \Delta \overset{=}{\Longrightarrow} [t/x]A_1\end{array}}{\Gamma; \Delta \overset{=}{\Longrightarrow} \exists x.\ A_1}\, \exists\mathrm{R}.$

We assumed $[\rho]A' = \exists x.\ A_1$, so $A' = \exists x.\ A_1'$ and $[\rho, t/a]([a/x]A_1') = [t/x]A_1$ for a new parameter a. Since a is new, $[\rho, t/a]\Gamma' = [\rho]\Gamma'$ and similarly for Δ', so we can apply the induction hypothesis to $\mathcal{D}_1$ to obtain $\mathcal{R}_1$ and $\mathcal{U}_1$ and construct

$$\frac{\begin{array}{c}\mathcal{R}_1 \\ \Gamma'; \Delta' \overset{=}{\Longrightarrow} [a/x]A_1' \setminus [a/x]F_1\end{array}}{\Gamma'; \Delta' \overset{=}{\Longrightarrow} \exists x.\ A_1' \setminus \exists x.\ F_1}\, \exists\mathrm{R}^a \quad \text{and} \quad \frac{\begin{array}{c}\mathcal{U}_1 \\ \models [\rho, t/a]([a/x]F_1)\end{array}}{\models [\rho]\exists x.\ F_1}\, \exists\mathrm{I}.$$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma; \Delta \overset{=}{\Longrightarrow} [a/x]A_1\end{array}}{\Gamma; \Delta \overset{=}{\Longrightarrow} \forall x.\ A_1}\, \forall\mathrm{R}^a.$

We assume $[\rho]A' = \forall x.\ A_1$, so $A' = \forall x.\ A_1'$ and $[\rho, a/a']([a'/x]A_1') = [a/x]A_1$ for an a' new in Γ', Δ' and $\forall x.\ A_1'$. We can then appeal to the induction hypothesis on $\mathcal{D}_1$ to obtain $\mathcal{R}_1$ and $\mathcal{U}_1$ and construct

$$\frac{\begin{array}{c}\mathcal{R}_1 \\ \Gamma'; \Delta' \overset{=}{\Longrightarrow} [a'/x]A_1' \setminus [a'/x]F_1\end{array}}{\Gamma'; \Delta' \overset{=}{\Longrightarrow} \forall x.\ A_1' \setminus \forall x.\ F_1}\, \forall\mathrm{I}^{a'} \quad \text{and} \quad \frac{\begin{array}{c}\mathcal{U}_1 \\ \models [\rho, a/a']([a'/x]F_1)\end{array}}{\models [\rho]\forall x.\ F_1}\, \forall\mathrm{I}^a.$$

□

Theorem 4.6 (Completeness of Equality Residuation) *If* $\Gamma; \Delta \overset{=}{\Longrightarrow} A$ *then* $\Gamma; \Delta \overset{=}{\Longrightarrow} A \setminus F$ *for some* F *and* $\models F$.

Proof: From Lemma 4.5 with $A' = A$, $\Delta' = \Delta$, $\Gamma' = \Gamma$, and ρ the identity substitution on the parameters in Γ, Δ, and A. □

Next we describe an algorithm for proving residuated formulas, that is, an algorithm for unification. We do this in two steps: first we solve the problem in the fragment without parameters and universal quantifiers and then we extend the solution to the general case.

There are numerous ways for describing unification algorithms in the literature. We describe the computation of the algorithm as the bottom-up search for the derivation of a judgment. We restrict the inference rules such that they are essentially deterministic, and the inference rules themselves can be seen as describing an algorithm. This algorithm is in fact quite close to the implementation of it in ML which is available together with these notes.[2]

In order to describe the algorithm in this manner, we need to introduce *existential variables* (often called *meta-variables* or *logic variables*) which are place-holders for the terms to be determined by unification. We use X to stand for existential variables.

The second concept we need is a *continuation*, which arises from the introduction rule for conjunction. This rule has two premises, which leaves the choice on how which premise to prove first when we work in a bottom-up fashion. Our algorithm commits to do the first conjunct first, but it has remember that the second conjunct remains to be proved. Equational formulas which have been postponed in this way are accumulated in the continuation, which is activated when there are no further equations to be solved. For now, a continuation is simply another formula denoted by S. Initially, we use $\top$ for S. Thus our main judgment describing the algorithm has the form "*F is satisfiable with continuation S*", written as $\models F \;/\; S$.

Continuations. The following rules introduce and manage the continuations.

$$\frac{\models F_1 \;/\; F_2 \wedge S}{\models F_1 \wedge F_2 \;/\; S}\wedge\mathrm{I} \qquad \frac{}{\models \top \;/\; \top}\top\mathrm{I}\top \qquad \frac{\models F \;/\; S}{\models \top \;/\; F \wedge S}\top\mathrm{I}\wedge$$

Existential Quantification. Existential variables are introduced for existential quantifiers. They must be new not only in F but also in S.

$$\frac{\models [X/x]F \;/\; S \quad X \text{ not in } F \text{ or } S}{\models \exists x.\, F \;/\; S}\exists\mathrm{I}$$

Despite the requirement on X to be new, the derivation of the premise is not parametric in X. That is, we cannot substitute an arbitrary term t for X in a derivation of the permiss and obtain a valid derivations, since the vr, rv, vv, and vv$'$ rules below require one or both sides of the equation to be an existential variable. Substituting for such a variables invalidates the application of these rules.

Predicate and Function Constants. An equation between the same function constant applied to arguments is decomposed into equations between the arguments. Unification fails if different function symbols are compared, but this

[2] [*perhaps at some point*]

is only indirectly reflected by an absence of an appropriate rule. Failure can also be explicitly incorporated in the algorithm (see Exercise **??**).

$$\frac{\models t_1 \doteq s_1 \wedge \cdots \wedge t_n \doteq s_n \;/\; S}{\models p(t_1, \ldots, t_n) \doteq p(s_1, \ldots, s_n) \;/\; S}\,\mathrm{pp} \qquad \frac{\models t_1 \doteq s_1 \wedge \cdots \wedge t_n \doteq s_n \;/\; S}{\models f(t_1, \ldots, t_n) \doteq f(s_1, \ldots, s_n) \;/\; S}\,\mathrm{rr}$$

These rules violate orthogonality by relying on conjunction in the premises for the sake of conciseness of the presentation. When f or p have no arguments, the empty conjunction in the premise should be read as $\top$.

Existential Variables. There are three rules for variables. We write r for terms of the form $f(t_1, \ldots, t_n)$. Existential variables always range over terms (and not propositions), so we do not need rules for equations of the form $X \doteq P$ or $P \doteq X$.

$$\frac{\models \top \;/\; [r/X]S \quad X \text{ not in } r}{\models X \doteq r \;/\; S}\,\mathrm{vr} \qquad \frac{\models \top \;/\; [r/X]S \quad X \text{ not in } r}{\models r \doteq X \;/\; S}\,\mathrm{rv}$$

These two rules come with the proviso that the existential variable X does not occur in the term t. This is necessary to ensure termination of these rules (when viewed as an algorithm) and to recognize formulas such as $\exists x.\, x \doteq f(x)$ as unprovable. This leaves equations of the form $X \doteq Y$ with to existential variables. We write two rules for this case to simplify the analysis.

$$\frac{\models \top \;/\; [Y/X]S}{\models X \doteq Y \;/\; S}\,\mathrm{vv} \qquad \frac{\models \top \;/\; S}{\models X \doteq X \;/\; S}\,\mathrm{vv}'$$

We now analyze these rules when viewed as an algorithm specification. First we observe that all rules have either no or one premise. Furthermore, for any judgment $\models F \;/\; S$ at most one rule is applicable, and in only one way (the choice of the new existential variable name X is irrelevant). Therefore these rules, when viewed as instructions for construction a derivation of a judgment $\models F \;/\; \top$ are deterministic, but may fail, in which case the formula is not provable.

Furthermore, the bottom-up search for a derivation of $\models F \;/\; S$ in this system will always terminate. The termination ordering involves five measures, ordered lexicographically as follows:

1. the number of free and quantified existential variables,
2. the number of predicate and function symbols,
3. the total number of logical symbols $\wedge$, $\top$, $\exists$ in F and S,
4. the number of logical symbols in F,
5. the number of equations.

This measure decreases in each rule:

$\wedge$I does not change (1)–(3) and decreases (4),

$\top$I$\top$ completes the search,

$\top$I$\wedge$ does not change (1)–(2) and decreases (3),

$\exists$I does not change (1)–(2) and decreases (3),

pp does not change (1) and decreases (2),

rr does not change (1) and decreases (2),

vr decreases (1) since X does not occur in r,

rv decreases (1) since X does not occur in r,

vv decreases (1), and

vv$'$ does not change (1)–(4) and decreases (5).

In some of these cases it is also possible that a measure of higher priority decreases (but never increases), preserving the strict decrease along the lexicographic ordering.

We also note that the continuation S is not completely general, but follows the grammar below.

$$\textit{Continuations} \quad S \quad ::= \quad \top \mid F \wedge S$$

In other words, it may be viewed as a stack of formulas. In the ML implementation, this stack is not represented explicitly. Instead we use the call stack of ML itself.

The desired soundness and completess theorems for this algorithm requires some generalizations based on substitutions for existential variables.

$$\textit{Ground Substitutions} \quad \theta \quad ::= \quad \cdot \mid \theta, t/X$$

We always assume that the terms t we assign to variables in substitutions do not contain existential variables. This assumption is reasonable, since we only use substitutions here to connect derivations for $\models F$ (which contains to existential variables) with derivations of $\models F' \mid S'$ (which contains existential variables).

Lemma 4.7 (Soundness Lemma for Unification) *If $\models F \mid S$ then there exists a ground substitution for the existential variables in F and S such that $\models [\theta]F$ and $\models [\theta]S$.*

Proof: By induction on the structure of $\mathcal{F} :: (\models F \mid S)$. $\square$

The soundness theorem follows easily from this lemma.

Theorem 4.8 (Soundness of Unification) *If $\models F \ / \ \top$ and F contains no existential variables, then $\models F$.*

Proof: From Lemma 4.7 for $S = \top$ and $\theta = \cdot$. □

Lemma 4.9 (Completeness Lemma for Unification) *If $\models F$ and $\models S$, then for any formulas F', continuations S' and substitutions θ for the existential variables in F' and S' such that $F = [\theta]F'$ and $S = [\theta]S'$ we have $\models F \ / \ S$.*

Proof: By nested inductions on $\mathcal{F} :: (\models F)$ and $\mathcal{S} :: (\models S)$. This means that when we appeal to the induction hypothesis on a subderivation of $\mathcal{F}$, $\mathcal{S}$ may be larger. We distinguish cases for $\mathcal{F}$.

Case: $\mathcal{F} = \dfrac{}{\models \top}\ \top\mathrm{I}$.

The we distinguish two subcases for $\mathcal{S}$. If $\mathcal{S}$ is $\top$I, the result is trivial by $\top$I$\top$. Otherwise

$$\mathcal{S} = \dfrac{\begin{array}{c}\mathcal{F}_1\\ \models F_1\end{array} \qquad \begin{array}{c}\mathcal{S}_2\\ \models S_2\end{array}}{\models F_1 \wedge S_2}\ \wedge\mathrm{I}$$

where $S = F_1 \wedge S_2$ for some F_1 and S_2. Then

$\mathcal{F}_1' :: (\models F_1 \ / \ S_2)$	By ind. hyp. on $\mathcal{F}_1$ and $\mathcal{S}_2$
$\mathcal{F}' :: (\models \top \ / \ F_1 \wedge S_2)$	By $\top$I$\wedge$

Case: $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1\\ \models F_1\end{array} \qquad \begin{array}{c}\mathcal{F}_2\\ \models F_2\end{array}}{\models F_1 \wedge F_2}\ \wedge\mathrm{I}$.

$\mathcal{F}_2' :: (\models F_2' \ / \ S')$	By ind. hyp. on $\mathcal{F}_2$ and $\mathcal{S}$
$\mathcal{S}_2 :: (\models F_2 \wedge S)$	By $\wedge$I from $\mathcal{F}_2$ and $\mathcal{S}$
$\mathcal{F}_1' :: (\models F_1' \ / \ F_2' \wedge S')$	By ind. hyp. on $\mathcal{F}_1$ and $\mathcal{S}_2$
$\mathcal{F}' :: (\models F_1' \wedge F_2' \ / \ S')$	By $\wedge$I from $\mathcal{F}_1'$.

Case: $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1\\ \models [t/x]F_1\end{array}}{\models \exists x.\ F_1}\ \exists\mathrm{I}$.

$F' = \exists x.\ F_1'$ and $[\theta](\exists x.\ F_1') = \exists x.\ F_1$	By assumption
$[\theta, t/X]([X/x]F_1') = [t/x]F_1$ for X not in F' or S'	
$[\theta, t/X]S' = S$	Since X is new
$\mathcal{F}_1' :: (\models [X/x]F_1' \ / \ S')$	By ind. hyp. on $\mathcal{F}_1$ and $\mathcal{S}$
$\mathcal{F} :: (\models \exists x.\ F_1' \ / \ S')$	By $\exists$I

Case: $\mathcal{F} = \dfrac{}{\models t \doteq t} \doteq\mathrm{I}$.

Here we proceed by an auxiliary induction on the structure of t. By assumption $[\theta]F' = (t \doteq t)$, so we have t' and t'' such that $[\theta]t' = [\theta]t'' = t$. We distinguish cases on t' and t'', showing three. The remaining ones are similar.

Subcase: $t' = f(t'_1, \ldots, t'_n)$ and $t'' = f(t''_1, \ldots, t''_n)$, so also $t = f(t_1, \ldots, t_n)$.

$\models t'_n \doteq t''_n \;/\; S'$	By ind. hyp. on t_n and $\mathcal{S}$
$\mathcal{S}_n :: (\models t_n \doteq t_n \wedge S)$	By $\wedge$I from $\doteq$I and $\mathcal{S}$
$\models t'_{n-1} \doteq t''_{n-1} \;/\; t'_n \doteq t''_n \wedge S'$	By ind. hyp. on t_{n-1} and $\mathcal{S}_n$.
$\models t'_1 \doteq t''_1 \;/\; t'_2 \doteq t''_2 \wedge \cdots \wedge t'_n \doteq t''_n \wedge S'$	As above
$\models t'_1 \doteq t''_1 \wedge t'_2 \doteq t''_2 \wedge \cdots \wedge t'_n \doteq t''_n \;/\; S'$	by $\wedge$I
$\models f(t'_1, \ldots, t'_n) \doteq f(t''_1, \ldots, t''_n) \;/\; S'$	by rr.

Subcase: $t' = X$ and $t'' = r$ but contains X. This is impossible, since we assumed $[\theta]t' = [\theta]t'' = t$.

Subcase: $t' = X$ and $t'' = r$ does not contain X. Then $[\theta]([r/X]S') = [\theta]S' = S$ since $[\theta]r = [\theta]X = t$ and θ is a ground substitution. By distinguishing cases for $\mathcal{S}$ as for $F = \top$ above, we conclude

$\models \top \;/\; [r/X]S'$	
$\models X \doteq r \;/\; S'$	By rule vr

□

The completeness theorem follows easily from this lemma.

Theorem 4.10 (Completeness of Unification) *If $\models F$ (where F contains no existential variables) then $\models F \;/\; \top$.*

Proof: From Lemma 4.9 with $S = \top$, $S' = \top$, $F' = F$ and $\theta = \cdot$. □

The generalization of the algorithm above to account for universal quantifiers and parameters is not completely straightforward. The difficulty is that $\forall x.\, \exists y.\, y \doteq x$ is valid, while $\exists y.\, \forall x.\, y \doteq x$ is not. We show an attempt to derive the latter which must be ruled out somehow.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\models \top \;/\; [a/Y]\top}\,\top\mathrm{I}\top}{\models Y \doteq a \;/\; \top}\,\mathrm{vr}}{\models \forall x.\, Y \doteq x \;/\; \top}\,\forall\mathrm{I}^a}{\models \exists y.\, \forall x.\, y \doteq x \;/\; \top}\,\exists\mathrm{I}$$

In this derivation, the application of $\top\mathrm{I}\top$ is correct since $[a/Y]\top = \top$. The problem lies in the fact a is new in the application of the $\forall\mathrm{I}^a$ rule, but only because we have not instantiated Y with a yet, which is necessary to complete the derivation.

There are two ways to solve this problem. More or less standard in theorem proving is *Skolemization* which we pursue in Exercise **??**. The dual solution notes for each existential variable which parameters may occur in its substitution term. In the example above, Y was introduced at a point where a did not yet occur, so the substitution of a for Y should be rejected.

In order to describe this concisely, we add a *parameter context* Ψ to the judgment which lists distinct parameters.

$$\textit{Parameter Context} \quad \Psi \quad ::= \quad \cdot \mid \Psi, a$$

This step is analogous to the localization of the hypotheses and should be considered merely a change in notation, not an essential change in the judgment itself. We annotate each judgment with the parameter context and introduce the new judmgnet "*t is closed with respect to Ψ*", written as $\Psi \models t\,\mathrm{term}$. It is defined by the following rules.

$$\frac{}{\Psi_1, a, \Psi_2 \vdash a\,\mathrm{term}}\,\mathrm{parm} \qquad \frac{\Psi \vdash t_1\,\mathrm{term} \quad \cdots \quad \Psi \vdash t_n\,\mathrm{term}}{\Psi \vdash f(t_1, \ldots, t_n)\,\mathrm{term}}\,\mathrm{root}$$

We modify the validity judgment for unification formulas to guarantee this condition.

$$\frac{\Psi \vdash t\,\mathrm{term} \qquad \Psi \models [t/x]F}{\Psi \models \exists x.\ F}\,\exists\mathrm{I} \qquad \frac{\Psi, a \models [a/x]F}{\Psi \models \forall x.\ F}\,\forall\mathrm{I}^a$$

When an existential variable X is introduced during the search for a derivation of a unification formula, we annotate it with the parameter context so we keep track of the admissible substitutions for X.

$$\frac{\Psi \models [X_\Psi/x]F \ /\ S \quad X_\Psi \text{ not in } F \text{ or } S}{\Psi \models \exists x.\ F \ /\ S}\,\exists\mathrm{I}$$

Parameters are introduced in the rule for universal quantifiers as before.

$$\frac{\Psi, a \models [a/x]F \ /\ S}{\Psi \models \forall x.\ F \ /\ S}\,\forall\mathrm{I}^a$$

An equation $X_\Psi \doteq t$ could now be solved immediately, if all parameters of t are contained in Ψ and X does not occur in t. However, there is one tricky case. Consider the judgment

$$a \models X_\cdot \doteq f(Y_a) \wedge Y_a \doteq a \ /\ \top$$

where X cannot depend on any parameters and Y can depend on a. This should have no solution, since $X_\cdot$ would have to be equal to $f(a)$, which is not permissible. On the other hand,

$$a \models X_\cdot \doteq f(Y_a) \wedge Y_a \doteq c \ /\ \top$$

for a constant c has a solution where Y_a is c and $X_{\cdot}$ is $f(c)$. So when we process an equation $X_\Psi = t$ we need to restrict any variable in t so it can depend only on the parameters in Ψ. In the example above, we would substitute $Y'_{\cdot}$ for Y_a.

In order to describe the algorithm, we internalize the judgment $\Psi \vdash t\,\mathit{term}$ as a new formula, written as $t\,|_\Psi$. We define it as follows.

$$\frac{\Psi' \models \top \;/\; S \quad \text{if } a \text{ in } \Psi}{\Psi' \models a\,|_\Psi /\; S}\,|\,\mathrm{a} \qquad \frac{\Psi' \models t_1\,|_\Psi \wedge \cdots \wedge t_n\,|_\Psi /\; S}{\Psi' \models f(t_1, \ldots, t_n)\,|_\Psi /\; S}\,|\,\mathrm{f}$$

$$\frac{\Psi' \models \top \;/\; [Y_{\Psi_2 \cap \Psi_1}/Y_{\Psi_2}]S}{\Psi' \models Y_{\Psi_2}\,|_{\Psi_1} /\; S}\,|\,\mathrm{v}$$

Here, $\Psi_1 \cap \Psi_2$ denotes the intersection of the two contexts. In the rules for variables, this is invoked as follows.

$$\frac{\Psi' \models r\,|_\Psi /\; [r/X_\Psi]S \quad \text{where } X_\Psi \text{ not in } r}{\Psi' \models X_\Psi \doteq r \;/\; S}\,\mathrm{vr}$$

$$\frac{\Psi' \models r\,|_\Psi /\; [r/X_\Psi]S \quad \text{where } X_\Psi \text{ not in } r}{\Psi' \models r \doteq X_\Psi \;/\; S}\,\mathrm{vr}$$

where r stands for a term $f(t_1, \ldots, t_n)$ or a parameter a. The variable rules are modified similarly.

$$\frac{\Psi' \models Y_{\Psi_2}\,|_{\Psi_1} /\; [Y_{\Psi_2}/X_{\Psi_1}]S}{\Psi' \models X_{\Psi_1} \doteq Y_{\Psi_2} \;/\; S}\,\mathrm{vv} \qquad \frac{\Psi' \models \top \;/\; S}{\Psi' x \models X_\Psi \doteq X_\Psi \;/\; S}\,\mathrm{vv}'$$

The use of continuations introduces on final complication. Consider the case of $(\forall x.\ F_1) \wedge F_2$. Since we linearize bottom-up search the parameter context Ψ will contain the parameter introduced for x when F_2 is finally considered after F_1 has been solved. This introduces spurious dependencies. To prohibit those, we build *closures* consisting of a formula and its parameter context on the continuation stack.

$$\text{Continuations} \quad S \quad ::= \quad \top \mid \{\Psi, F\} \wedge S$$

The rules for continuations are modified as follows.

$$\frac{\Psi \models F_1 \;/\; \{\Psi, F_2\} \wedge S}{\Psi \models F_1 \wedge F_2 \;/\; S}\,\wedge\mathrm{I} \qquad \frac{}{\Psi \models \top \;/\; \top}\,\top\mathrm{I}\top \qquad \frac{\Psi \models F \;/\; S}{\Psi' \models \top \;/\; \{\Psi, F\} \wedge S}\,\top\mathrm{I}\wedge$$

The termination argument is only slightly more difficult, since the restriction operation is a structural recursion over the term r and does not increase the number of variables or equations.

The soundness and completeness theorems from above extend to the problem with parameters, but become more difficult. The principal new notion we need is an *admissible substitution* θ which has the property that for every existential variable X_Ψ, $\Psi \vdash [\theta]X_\Psi$ term (see Exercise **??**).

The ML implementation takes advantage of the fact that whenever a variable must be restricted, one of the two contexts is a prefix of the other. This is because every equation in a formula F lies beneath a path of possibly alternating quantifiers, a so-called *mixed quantifier prefix*. When we apply the rules above algorithmically, we instantiate each existentially quantified variable with a new free existential variable which depends on all parameters which were introduced for the universally quantified variables to its left. Clearly, then, for any two variables in the same equation, one context is a prefix of the other. Our ML implementation does take advantage of this observation by simplifying the intersection operation.

We can take this optimization a step further and only record with an integer (a kind of time stamp), which parameters an existential variable may depend on. This improves the efficiency of the algorithm even further, since we only need to calculate the minimum of two integers instead of intersecting two contexts during restriction. In the ML code for this class, we did not optimize to this extent.

Chapter 5

Linear Logic Programming

When we think of logic we generally first consider it as a discipline concerned with the study of propositions, truth, and inference. This may appear at first to be independent from any notion of computation. However, there are two immediate connections: *proofs as programs* and *proof search as computation*.

In constructive logic (and our approach has been constructive) we can view a proof as defining a construction (algorithm, program). For example, a proof of $A \supset B$ shows how to achieve goal B when given the resource A. Carrying out such a construction when we actually have obtained resource A the corresponds to computation. This notion of computation is most closely related to functional programming, but because of the state-aware nature of linear logic it also has some imperative flavor. We will discuss a computational interpretation of linear logic along these lines in Chapter 6.

Another computational interpretation is closer to the way we have been using linear logic so far. Reconsider, for example, the encoding of Petri nets in linear logic. Each possible computation step of the Petri net is modeled by a corresponding inference step in linear logic. As a result, reachability in a Petri net corresponds to provability in its linear encoding. More importantly, each possible computation of a Petri net corresponds to a proof, and carrying out a computation corresponds to the construction of a proof. In other words, proof search in linear logic corresponds to computation.

This leads to the question if we can exploit this correspondence in order to design a *programming language* based on linear logic where computation is indeed proof search. The result of computation then is a particular proof, or possibly a collection or enumeration of proofs depending on the characteristics of the language. A program in this setting is simply a collection of propositions that, through their form, will lead the proof search engine down a particular path, thereby achieving a particular computation. In order to make this both feasible from the point of view of an implementation and predictable to the programmer, we need to full linear logic. We would like to emphasize that even on this fragment (called LHHF for Linear Hereditary Harrop Formulas), not every specification is executable, nor is it intended to be. We hope the development and the examples in this chapter will clarify this point.

5.1 Logic Programming as Goal-Directed Search

Our first approach to logic programming is via the notion of *goal-directed search*. It turns out that this view diverges from our earlier examples because it does not incorporate any concurrency. However, some of our earlier encodings can be rewritten to fit into the language given below.

Assume we are trying to prove $\Gamma; \Delta \Longrightarrow A$ where Γ are the unrestricted hypotheses (which correspond to the program), Δ are the linear hypotheses (which correspond to current state) and A which corresponds to the goal. The idea of goal-directed search is that we always first break down the structure of the goal A until it is atomic (P). At that point we focus on one of the hypotheses in Γ or Δ and apply consecutive left rules until the focus formula matches P and we can solve the subgoals generated during this process. This phase of the computation corresponds to a procedure call, where the generated subgoals correspond to the procedure body which is then executed in turn.

In order to have a satisfactory logical interpretation of the program (in addition to the computational one above), we would like this search procedure to be sound and non-deterministically complete. Soundness simply means that we only find valid proof, which is easy to accomplish since we only restrict the applications of ordinary sequent rules. Completeness means that for every derivation for the original judgment there is a sequence of choices according to the strategy above which finds this derivation. As we have seen in the development of focusing (Section 4.2), the goal-directed strategy above is generally *not* complete. However, it is complete if we restrict ourselves to right asynchronous connectives, because focusing is complete and all the connectives are orthogonal to each other.

This fragment (with some irrelevant, minor deviations) is called the system of linear hereditary Harrop formulas (LHHF).

$$A \quad ::= \quad P \mid A_1 \multimap A_2 \mid A_1 \& A_2 \mid \top \mid A_1 \supset A_2 \mid \forall x.\, A$$

We obtain the foundation for its operational semantics simply from the focusing system, restricted to the above connectives. Since they are all right asynchronous, we only need two of the four judgments. We call this a system of *uniform proofs* [MNPS91]. For the case of linear, this system has first been proposed by Hodas and Miller [HM94, Hod94], although similar ideas for classical linear logic had been developed independently by Andreoli [AP91, And92].

Goal-Directed Search. This corresponds to the *inversion* phase of the focusing system. Because all right asynchronous propositions are left synchronous, we do not need to collect them into an ordered context Ω but add them directly to the left synchronous hypotheses in Δ. We write

$$\Gamma; \Delta \Longrightarrow A \Uparrow$$

From the focusing system we obtain the following rules.

$$\frac{\Gamma; \Delta, A \Longrightarrow B \Uparrow}{\Gamma; \Delta \Longrightarrow A \multimap B \Uparrow} \multimap R \qquad \frac{\Gamma; \Delta \Longrightarrow A \Uparrow \qquad \Gamma; \Delta \Longrightarrow B \Uparrow}{\Gamma; \Delta \Longrightarrow A \& B \Uparrow} \& R$$

$$\frac{}{\Gamma; \Delta \Longrightarrow \top \Uparrow} \top R \qquad \frac{\Gamma, A; \Delta \Longrightarrow B \Uparrow}{\Gamma; \Delta \Longrightarrow A \supset B \Uparrow} \supset R$$

$$\frac{\Gamma; \Delta \Longrightarrow [a/x]A \Uparrow}{\Gamma; \Delta \Longrightarrow \forall x.\, A \Uparrow} \forall R^a$$

Procedure Call. This corresponds to the *focusing* phase of the focusing system. In this case we can carry over the judgment directly

$$\Gamma; \Delta; A \Downarrow \Longrightarrow P$$

where A is the focus formula and P is always atomic. This judgment is also called *immediate entailment* because A must decompose to directly yield P as can be seen from the rules below.

This phase is triggered by a decision, if the goal formula is atomic.

$$\frac{\Gamma; \Delta; A \Downarrow \Longrightarrow P}{\Gamma; \Delta, A \Longrightarrow P \Uparrow} \mathsf{decideL} \qquad \frac{\Gamma, A; \Delta; A \Downarrow \Longrightarrow P}{\Gamma, A; \Delta \Longrightarrow P \Uparrow} \mathsf{decideL!}$$

During this phase, we simply carry out the left rules on the focus formula. Since we can never obtain a left asynchronous proposition (there are none among the linear hereditary Harrop formulas), we can only succeed if we obtain an atomic proposition equal to P.

$$\frac{}{\Gamma; \cdot; P \Downarrow \Longrightarrow P} \mathsf{init}$$

$$\frac{\Gamma; \Delta_1; B \Downarrow \Longrightarrow P \qquad \Gamma; \Delta_2 \Longrightarrow A \Uparrow}{\Gamma; \Delta_1, \Delta_2; A \multimap B \Downarrow \Longrightarrow P} \multimap L$$

$$\frac{\Gamma; \Delta; A \Downarrow \Longrightarrow P}{\Gamma; \Delta; A \& B \Downarrow \Longrightarrow P} \& L_1 \qquad \frac{\Gamma; \Delta; B \Downarrow \Longrightarrow P}{\Gamma; \Delta; A \& B \Downarrow \Longrightarrow P} \& L_2$$

no left rule for $\top$ $$\frac{\Gamma; \Delta; B \Downarrow \Longrightarrow P \qquad \Gamma; \cdot \Longrightarrow A \Uparrow}{\Gamma; \Delta; A \supset B \Downarrow \Longrightarrow P} \supset L$$

$$\frac{\Gamma; \Delta; [t/x]A \Downarrow \Longrightarrow P}{\Gamma; \Delta; \forall x.\, A \Downarrow \Longrightarrow P} \forall L$$

Some of the premises of these rules refer back to goal-directed search. The collection of these premises for a particular focusing step (that is, procedure call) corresponds to the procedure body. Operationally, they will be solved only after we know if the init rule applies at the end of the sequence of focusing steps.

It is easy to see that uniform proofs are sound and complete with respect to the sequent calculus via the soundness and completeness of focusing.[1]

5.2 An Operational Semantics

The system of uniform proofs from the previous section is only the basis of an actual operational semantics for LHHF. There are still a number of choices left and we have to specify how they are resolved in order to know precisely how a query

$$\Gamma; \Delta \Longrightarrow A \Uparrow$$

executes. We organize this discussion into the forms of the non-deterministic choices that remain. We are not formal here, even though a formal description can certainly be given.[2]

Existential Non-Determinism. This arises in the choice of the term t in the $\forall$L rule during the focusing phase. This is resolved by substituting instead a logic variable X, where it is explicitly remember which parameters a the variable X may depend on. For initial sequents

$$\frac{}{\Gamma; \cdot; P \Downarrow \Longrightarrow P}\,\mathsf{init}$$

we instead allow the hypothesis P and goal P' and unify them instead of testing them for equality. Since we can always find a most general unifier, this involves no unnecessary overcommitment and we do not have to backtrack in order to retain completeness.

Conjunctive Non-Determinism. If several subgoals arise during focusing, or because we have a goal $A_1 \& A_2$, we have to solve all subgoals but the order presents a form of conjunctive non-determinism. We resolve this by always solving the premises in the uniform proof rules from left to right. This has the desirable effect that we only attempt to solve a subgoal once we have unified the atomic goal P with the proposition P' at the end of the focus formula.

Disjunctive Non-Determinism. Disjunctive non-determinism arises in the choice of the focus formula once the goal is atomic, and in the choice of the left rule if the focus formula is an alternative conjunction. This is resolved via depth-first search and backtracking. For the decision how to focus, we use the following order:

[1] *[add more formal statement]*
[2] *[several citations]*

1. First the linear or unrestricted hypotheses that were introduced during proof search, where we try the most recently made assumption first (right-to-left, in our notation).

2. Then we try the unrestricted assumptions that were fixed at the beginning of the search (the program), trying the propositions from first to last (left-to-right in our notation).

For alternative conjunctions as the focus formula, we first try the left conjunct and then the right conjunct.

Resource Non-Determinism. Resource non-determinism arises in the $\multimap L$ rule, where we have to split assumptions between the premises. Conceptually, this can be resolved by introducing Boolean constraints [HP97] and solving them eagerly. In order to gain a better intuition what this means operationally, equivalent systems that avoid explicit creation of constraints have been developed [CHP00]. We will give some intuition in Section 5.3 where we introduce the *input/output model* for resource management.

Unfortunately, the way we treate disjunctive non-determinism via depth-first search and backtracking means that there may be proofs we never find because the interpreter following our operational semantics does not terminate. Many attempts have been made to alleviate this difficulty, but none are entirely satisfactory. Depth-first search seems to be critical to obtain a simple and understandable operational semantics for programs that allows algorithms to be implemented efficiently in a logic programming language.

Even though the interpreter is incomplete in this sense, the non-deterministic completeness of the uniform proof system is still very important. This is because we would like to be able to interpret failure as unprovability. Since the uniform proof system is non-deterministically complete, we know that if the interpreter fails finitely and reports no proof can be found because all choices have been exhausted, then there cannot be a proof of the original goal.

To summarize, the interpreter may exhibit three behaviors.

1. Succeed with a proof. By soundness, the goal is provable. If we backtrack further, we may get other proofs.

2. Fail. By non-deterministic completeness, the goal is unprovable and hence not true in general.

3. Run. In this case we have no information yet. We cannot observe if the interpreter will run forever, so we have to let it run and hope for eventual termination, either with success or failure.

Note that these are exactly the same even if our interpreter were complete in the strong sense. The only difference would be that if there is a proof, the running interpreter would eventually succeed. It cannot always fail if there is no proof, because of the undecidability of this fragment (which is easy to verify, see Exerciseexc:lhhf-undec). One should note, however, that even simple decidable fragments may exhibit non-terminating behavior.

This observation reminds us that linear logic programming does not provide a general theorem prover. It is not possible to write an arbitrary specification (even in the LHHF fragment) and obtain a reasonable program. Instead, it is often possible to write programs at a very high level of abstraction, and sometimes even possible to few specification directly as programs, but just as often this is not the case. Some examples below should help to clarify this.

5.3 Deterministic Resource Management

In order to use linear hereditary Harrop formulas effectively as a logic programming language, we need to understand how resource non-determinism is resolved. This can be understood in three stages—we give here only the first and most important one.

The only rule where we have to consider how to split resources is the $\multimap$L rule.

$$\frac{\Gamma; \Delta_1; B \Downarrow \Longrightarrow P \qquad \Gamma; \Delta_2 \Longrightarrow A \Uparrow}{\Gamma; \Delta_1, \Delta_2; A \multimap B \Downarrow \Longrightarrow P} \multimap \mathrm{L}$$

Note that the left premise will be solved first and then the right premise. The way we resolve this choice is that we pass all the resources to the left premise and keep track which ones were actually needed in the proof of $B \Downarrow \Longrightarrow P$. The remaining ones are passed to the second premise once the first premise has succeeded. This strategy only make sense because we have already committed to solve conjunctive non-determinism by left-to-right subgoal selection.

This describes the *input/output model* of resource management. We reuse some of the notation introduced to describe Boolean constraints, by writing $u{:}A[1]$ for a linear hypothesis that is there, and $u{:}A[0]$ for a linear hypothesis that has been consumed somewhere. No other Boolean expression arise here. The main judgments are now

$$\begin{array}{c} \Gamma; \Delta_I \backslash \Delta_O \Longrightarrow A \Uparrow \\ \Gamma; \Delta_I \backslash \Delta_O; A \Downarrow \Longrightarrow P \end{array}$$

where Δ_I stands for the input resources that may be consumed and Δ_O stands for the output resources that were not consumed. Note that the judgment is hypothetical in Δ_I but not in Δ_O. First, the goal-directed phase of search.

$$\frac{\Gamma; \Delta_I, u{:}A[1] \backslash \Delta_O, u{:}A[0] \Longrightarrow B \Uparrow}{\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow A \multimap B \Uparrow} \multimap\mathrm{R}$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow A \Uparrow \qquad \Gamma; \Delta_I \backslash \Delta_O \Longrightarrow B \Uparrow}{\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow A \& B \Uparrow} \&\mathrm{R}$$

$$\frac{\Delta_I \supseteq \Delta_O}{\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow \top \Uparrow} \top\mathrm{R} \qquad \frac{\Gamma, A; \Delta_I \backslash \Delta_O \Longrightarrow B \Uparrow}{\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow A \supset B \Uparrow} \supset\mathrm{R}$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow [a/x]A \Uparrow}{\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow \forall x.\ A \Uparrow} \forall\mathrm{R}^a$$

The right rule for linear implication requires that the linear assumption be consumed ($A[0]$). For $\top$ we have to allow an arbitrary subset of the input hypotheses to be consumed. This relation is defined by

$$\frac{}{\cdot \supseteq \cdot} \qquad \frac{\Delta_I \supseteq \Delta_O}{\Delta_I, u{:}A[0] \supseteq \Delta_O, u{:}A[0]}$$

$$\frac{\Delta_I \supseteq \Delta_O}{\Delta_I, u{:}A[1] \supseteq \Delta_O, u{:}A[1]} \qquad \frac{\Delta_I \supseteq \Delta_O}{\Delta_I, u{:}A[1] \supseteq \Delta_O, u{:}A[0]}$$

The non-determinism that arises in this rule has to be eliminated in a second step (see [CHP00]).[3]

Second, the transition between the phases. Note that linear hypotheses are consumed here, and not at the rules for initial sequents.

$$\frac{\Gamma; \Delta_I \backslash \Delta_O; A \Downarrow \Longrightarrow P}{\Gamma; \Delta_I, u{:}A[1] \backslash \Delta_O, u{:}A[0] \Longrightarrow P \Uparrow} \mathsf{decideL} \qquad \frac{\Gamma, A; \Delta_I \backslash \Delta_O; A \Downarrow \Longrightarrow P}{\Gamma, A; \Delta_I \backslash \Delta_O \Longrightarrow P \Uparrow} \mathsf{decideL!}$$

Third, the focusing phase. The critical rule is the one for $\multimap$L where resources are passed through from left to right with Δ_M representing the hypotheses that have not been consumed in the proof of the first premise. Also note that for initial sequent we simply pass through all linear hypothesis rather than checking if they are empty. This is because we are no longer required, locally, that all linear assumptions are used, since some pending subgoal may consume it later.

[3] [*maybe reformulate and add here*]

$$\frac{}{\Gamma; \Delta_I \backslash \Delta_I; P \Downarrow \Longrightarrow P}\,\textsf{init}$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_M; B \Downarrow \Longrightarrow P \qquad \Gamma; \Delta_M \backslash \Delta_O \Longrightarrow A \Uparrow}{\Gamma; \Delta_I \backslash \Delta_O; A \multimap B \Downarrow \Longrightarrow P} \multimap \mathrm{L}$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_O; A \Downarrow \Longrightarrow P}{\Gamma; \Delta_I \backslash \Delta_O; A \& B \Downarrow \Longrightarrow P}\,\&\mathrm{L}_1 \qquad \frac{\Gamma; \Delta_I \backslash \Delta_O; B \Downarrow \Longrightarrow P}{\Gamma; \Delta_I \backslash \Delta_O; A \& B \Downarrow \Longrightarrow P}\,\&\mathrm{L}_2$$

$$\text{no left rule for } \top \qquad \frac{\Gamma; \Delta_I \backslash \Delta_O; B \Downarrow \Longrightarrow P \qquad \Gamma; \cdot \Longrightarrow A \Uparrow}{\Gamma; \Delta_I \backslash \Delta_O; A \supset B \Downarrow \Longrightarrow P} \supset\mathrm{L}$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_O; [t/x]A \Downarrow \Longrightarrow P}{\Gamma; \Delta_I \backslash \Delta_O; \forall x.\, A \Downarrow \Longrightarrow P}\,\forall\mathrm{L}$$

In order to execute a query $\Gamma; \Delta \Longrightarrow A$ we instead execute

$$\Gamma; \Delta[1] \backslash \Delta[0] \Longrightarrow A \Uparrow$$

where $(u_1{:}A_1, \ldots, u_n{:}A_n)[b]$ is shorthand for $u_1{:}A_1[b], \ldots, u_n{:}A_n[b]$. This guarantees that all linear hypotheses have indeed be consumed.

In the statement of soundness and completeness, however, we need to be more general to account for intermediate states. The idea of soundness is that if $\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow A \Uparrow$ then if we delete all hypotheses from Δ_I that are still in Δ_O, we should have a uniform proof from the resulting hypotheses. We therefore define subtraction, $\Delta_I - \Delta_O = \Delta$, where Δ_I and Δ_O have Boolean annotations and Δ does not.

$$\frac{}{\cdot - \cdot = \cdot} \qquad \frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, u{:}A[1]) - (\Delta_I, u{:}A[0]) = (\Delta, u{:}A)}$$

$$\frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, u{:}A[1]) - (\Delta_I, u{:}A[1]) = \Delta} \qquad \frac{\Delta_I - \Delta_O = \Delta}{(\Delta_I, u{:}A[0]) - (\Delta_I, u{:}A[0]) = \Delta}$$

We can then prove soundness directly.

Theorem 5.1 (Soundness of I/O Resource Management)

1. *If* $\Gamma; \Delta_I \backslash \Delta_O \Longrightarrow A \Uparrow$ *then* $\Delta_I - \Delta_O = \Delta$ *and* $\Gamma; \Delta \Longrightarrow A \Uparrow$

2. *If* $\Gamma; \Delta_I \backslash \Delta_O; A \Downarrow \Longrightarrow P$ *then* $\Delta_I - \Delta_O = \Delta$ *and* $\Gamma; \Delta; A \Downarrow \Longrightarrow P$

Proof: By mutual induction on the structure of the given derivation.[4] □

[4] [*check for lemmas and write out some cases*]

For the completness direction we need to generalize the induction hypothesis somewhat differently.

Theorem 5.2 (Completeness of I/O Resource Management)

1. *If* $\Gamma;\Delta \Longrightarrow A \Uparrow$ *then* $\Gamma;\Delta[1],\Delta_O \backslash \Delta[0],\Delta_O \Longrightarrow A \Uparrow$ *for any* Δ_O.
2. *If* $\Gamma;\Delta;A \Downarrow \Longrightarrow P$ *then* $\Gamma;\Delta[1],\Delta_O \backslash \Delta[0],\Delta_O; A \Downarrow \Longrightarrow P$ *for andy* Δ_O.

Proof: By mutual induction on the structure of the given derivation.[5] □

5.4 Some Example Programs

We start with some simple programs. Following the tradition of logic programming, we write implications in the program (Γ) in reverse so that $A \circ\!- B$ means $B \multimap A$. Implication in this direction is left associative, and subgoals solved (visually) from left-to-right. So,

$$P \circ\!- Q \circ\!- R$$

stands for $(P \circ\!- Q) \circ\!- R$ which is the same as $R \multimap (Q \multimap P)$. If P matches the current atomic goal, then first subgoal to be solved is Q and then R. This is consistent with the informal operational semantics explained above.

The first program is non-terminating for the simple query p.

$u_1 \;:\; p \circ\!- p.$
$u_0 \;:\; p.$

Then a goal $\Longrightarrow p$ under this program will diverge, since it will use u_1 first, which produces the identical subgoal of $\Longrightarrow p$. If we reorder the clauses

$u_0 \;:\; p.$
$u_1 \;:\; p \circ\!- p.$

the query $\Longrightarrow p$ will produce the immediate proof (u_0) first and, if further answer are requested, succeed arbitrarily often with different proofs. We can slightly complicate this example by adding an argument to p.

$u_0 \;:\; p(0).$
$u_s \;:\; \forall x.\, p(\mathsf{s}(x)) \circ\!- p(x).$

In a query we can now leave an existential variable, indicated by an uppercase letter, $\Longrightarrow p(X)$. this query will succeed and print the answer substitution $X = 0$. If further solutions are requested, the program will enumerate $X = \mathsf{s}(0)$, $X = \mathsf{s}(\mathsf{s}(0))$, etc. In general, most logic programming language implementation print only substitutions for existential variables in a query, but not other aspects of the proof it found.

The trivial examples above do not take advantage of the expressive power of linear logic and could equally well be written, for example, in Prolog.

For the next example we introduce lists as terms, using constructors nil and cons. For example, the list $1, 2, 3$ would be written as $\mathsf{cons}(1, \mathsf{cons}(2, \mathsf{cons}(3, \mathsf{nil})))$. A program to enumerate all permutations of a list is the following.

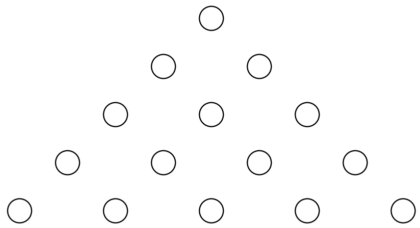
[5] [*check for lemmas and write out some cases*]

$$
\begin{array}{lcl}
p_0 & : & \mathsf{perm}(\mathsf{cons}(X, L), K) \mathrel{\circ\!\!-} (\mathsf{elem}(X) \multimap \mathsf{perm}(L, K)) \\
p_1 & : & \mathsf{perm}(\mathsf{nil}, \mathsf{cons}(X, K)) \mathrel{\circ\!\!-} \mathsf{elem}(X) \mathrel{\circ\!\!-} \mathsf{perm}(\mathsf{nil}, K) \\
p_2 & : & \mathsf{perm}(\mathsf{nil}, \mathsf{nil})
\end{array}
$$

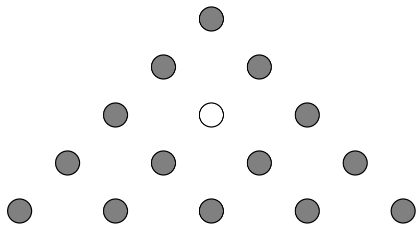
Here we have left universal quantifiers on X, L, and K implicit in each declaration in order to shorten the program. This is also supported by implementations of logic programming languages.

We assume a query of the form $\Longrightarrow \mathsf{perm}(l, K)$ where l is a list and K is a free existential variable. The program iterates over the list l with p_0, creating a linear hypothesis $\mathsf{elem}(t)$ for every element t of the list. Then it repeatedly uses clause p_1 to consume the linear hypothesis in the output list K. When there are no longer any linear hypotheses, the last clause p_2 will succeed and therefore the whole program.

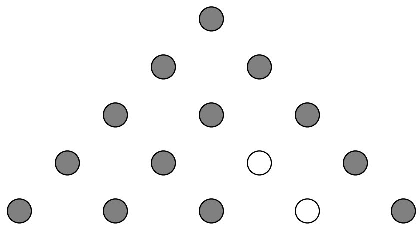
As a second example we consider a simple solitaire game with a board of the form



Each circle represents a hole which is filled if that hole contains a peg. The initial position



has one empty hole, while all other holes are filled by pegs (14 all together). We move by jumping one peg over another if the hole behind it is empty. For example, in the given initial position there are only two legal moves, one of which leads to the following situation:



The goal is to achieve a configuration in which only one peg is left. Alternatively, we can say the goal is to make 13 consecutive moves starting from the original position.

We use two digits to represent the address of each hole on the board in the following manner:

				00				
			10		11			
		20		21		22		
	30		31		32		33	
40		41		42		43		44

To represent the current situation, we have two predicates

$\mathsf{empty}(x, y)$	Hole xy is empty
$\mathsf{peg}(x, y)$	Hold xy contains a peg

Then the initial situation is represent by the linear hypotheses

$$\begin{array}{l}
\mathsf{peg}(0,0), \\
\mathsf{peg}(1,0), \mathsf{peg}(1,1), \\
\mathsf{peg}(2,0), \mathsf{empty}(2,1), \mathsf{peg}(2,2), \\
\mathsf{peg}(3,0), \mathsf{peg}(3,1), \mathsf{peg}(3,2), \mathsf{peg}(3,4), \\
\mathsf{peg}(4,0), \mathsf{peg}(4,1), \mathsf{peg}(4,2), \mathsf{peg}(4,3), \mathsf{peg}(4,4).
\end{array}$$

Now each possible move can be represented in the style of our earlier encodings as a linear implication. Because there are six possible directions for jumping (although many are impossible for any given peg), we have six different rules. We name each rule with its compass direction

$$\begin{array}{rcl}
sw & : & \mathsf{peg}(x,y) \otimes \mathsf{peg}(x+1,y) \otimes \mathsf{empty}(x+2,y) \\
& & \multimap \mathsf{empty}(x,y) \otimes \mathsf{empty}(x+1,y) \otimes \mathsf{peg}(x+2,y) \\
ne & : & \mathsf{peg}(x+2,y) \otimes \mathsf{peg}(x+1,y) \otimes \mathsf{empty}(x,y) \\
& & \multimap \mathsf{empty}(x+2,y) \otimes \mathsf{empty}(x+1,y) \otimes \mathsf{peg}(x,y) \\
e & : & \mathsf{peg}(x,y) \otimes \mathsf{peg}(x,y+1) \otimes \mathsf{empty}(x,y+2) \\
& & \multimap \mathsf{empty}(x,y) \otimes \mathsf{empty}(x,y+1) \otimes \mathsf{peg}(x,y+2) \\
w & : & \mathsf{peg}(x,y+2) \otimes \mathsf{peg}(x,y+1) \otimes \mathsf{empty}(x,y) \\
& & \multimap \mathsf{empty}(x,y+2) \otimes \mathsf{empty}(x,y+1) \otimes \mathsf{peg}(x,y) \\
se & : & \mathsf{peg}(x,y) \otimes \mathsf{peg}(x+1,y+1) \otimes \mathsf{empty}(x+2,y+2) \\
& & \multimap \mathsf{empty}(x,y) \otimes \mathsf{empty}(x+1,y+1) \otimes \mathsf{peg}(x+2,y+2) \\
nw & : & \mathsf{peg}(x+2,y+2) \otimes \mathsf{peg}(x+1,y+1) \otimes \mathsf{empty}(x,y) \\
& & \multimap \mathsf{empty}(x+2,y+2) \otimes \mathsf{empty}(x+1,y+1) \otimes \mathsf{peg}(x,y)
\end{array}$$

In order to specify the goal, we can specify the precise desired configuration. In this example we see if we won by counting the number of moves, so we can add $\mathsf{count}(0)$ to the initial state, $\mathsf{count}(n)$ to the left-hand side of every implication and $\mathsf{count}(n+1)$ to the right-hand side. To solve the puzzle we then have to prove

$$\Gamma_0; \Delta_0, \mathsf{count}(0) \Longrightarrow \mathsf{count}(13) \otimes \top$$

where Γ_0 is the program above and Δ_0 is the representation of the initial situation.

This representation is adequate, but unfortunately it does not fall within the class of linear hereditary Harrop formulas because of its use of $\otimes$. In fact, if we look at the seqent we have to prove above to solve the puzzle, the proof will not proceed in a goal-directed fashion. Instead, we have to continually focus on propositions in Γ_0 until the goal happens to be provable in the end.

Fortunately, we can transfer it into the class LHHF by taking advantage of two ideas. The first is a logical law called *uncurrying*:

$$(A \otimes B) \multimap C \dashv\vdash A \multimap (B \multimap C)$$

This allows us to eliminate simultanous conjunction on the left-hand side of linear implications. But what about the right-hand side? In this case no similar local transformation exists. Instead we transform the whole program by using a so-called A-translation or continuation-passing transform.[6] Normally, a move $A \multimap B$ is a function that transforms the resource A into the goal B. However, instead of returning B, we will now add a second argument that consumes the result B and eventually returns a final answer. By the substitution principle (or cut, in the sequent calculus), this is sound and complete. So $A \multimap B$ is transformed into $A \multimap (B \multimap p) \multimap p$ where p is a new atomic predicate. Note that this shifts B to the left-hand side of an implication, so we can now apply uncurrying in case it is a tensor. In general, all clauses of the program need to be transformed with the same new propositional parameter or constant p.

If we read the transformed proposition operationally (also changing the order of subgoals),

$$p \circ\!- A \circ\!- (B \multimap p),$$

it means that in order to prove p we have to prove A (which may consume some resources), then assume B and make a recursive call. In our example, we call the new predicate *jump* and the first clause

$$\begin{array}{rcl} sw & : & \mathsf{peg}(x,y) \otimes \mathsf{peg}(x+1,y) \otimes \mathsf{empty}(x+2,y) \\ & & \multimap \mathsf{empty}(x,y) \otimes \mathsf{empty}(x+1,y) \otimes \mathsf{peg}(x+2,y) \end{array}$$

is transformed into

$$\begin{array}{rcl} sw & : & \mathsf{jump} \circ\!- (\mathsf{peg}(x,y) \otimes \mathsf{peg}(x+1,y) \otimes \mathsf{empty}(x+2,y)) \\ & & \circ\!- (\mathsf{empty}(x,y) \otimes \mathsf{empty}(x+1,y) \otimes \mathsf{peg}(x+2,y) \multimap \mathsf{jump}) \end{array}$$

and similarly for the other clauses. Now we can eliminate the simultaneous conjunction by uncurrying to obtain an equivalent proposition in the LHHF

[6] [*add citations*]

fragment.

$$\begin{array}{rcl} sw & : & \mathsf{jump} \circ\!- \mathsf{peg}(x, y) \circ\!- \mathsf{peg}(x+1, y) \circ\!- \mathsf{empty}(x+2, y) \\ & & \circ\!-(\mathsf{empty}(x, y) \multimap \mathsf{empty}(x+1, y) \multimap \mathsf{peg}(x+2, y) \multimap \mathsf{jump}) \end{array}$$

In order to count the number of moves we just add an argument to the jump predicate that we count down to zero.

$$\begin{array}{rcl} sw & : & \mathsf{jump}(n+1) \circ\!- \mathsf{peg}(x, y) \circ\!- \mathsf{peg}(x+1, y) \circ\!- \mathsf{empty}(x+2, y) \\ & & \circ\!-(\mathsf{empty}(x, y) \multimap \mathsf{empty}(x+1, y) \multimap \mathsf{peg}(x+2, y) \multimap \mathsf{jump}(n)) \\ ne & : & \mathsf{jump}(n+1) \circ\!- \mathsf{peg}(x+2, y) \circ\!- \mathsf{peg}(x+1, y) \circ\!- \mathsf{empty}(x, y) \\ & & \circ\!-(\mathsf{empty}(x+2, y) \multimap \mathsf{empty}(x+1, y) \multimap \mathsf{peg}(x, y) \multimap \mathsf{jump}(n)) \\ e & : & \mathsf{jump}(n+1) \circ\!- \mathsf{peg}(x, y) \circ\!- \mathsf{peg}(x, y+1) \circ\!- \mathsf{empty}(x, y+2) \\ & & \circ\!-(\mathsf{empty}(x, y) \multimap \mathsf{empty}(x, y+1) \multimap \mathsf{peg}(x, y+2) \multimap \mathsf{jump}(n)) \\ w & : & \mathsf{jump}(n+1) \circ\!- \mathsf{peg}(x, y+2) \circ\!- \mathsf{peg}(x, y+1) \circ\!- \mathsf{empty}(x, y) \\ & & \circ\!-(\mathsf{empty}(x, y+2) \multimap \mathsf{empty}(x, y+1) \multimap \mathsf{peg}(x, y) \multimap \mathsf{jump}(n)) \\ se & : & \mathsf{jump}(n+1) \circ\!- \mathsf{peg}(x, y) \circ\!- \mathsf{peg}(x+1, y+1) \circ\!- \mathsf{empty}(x+2, y+2) \\ & & \circ\!-(\mathsf{empty}(x, y) \multimap \mathsf{empty}(x+1, y+1) \multimap \mathsf{peg}(x+2, y+2) \multimap \mathsf{jump}(n)) \\ nw & : & \mathsf{jump}(n+1) \circ\!- \mathsf{peg}(x+2, y+2) \circ\!- \mathsf{peg}(x+1, y+1) \circ\!- \mathsf{empty}(x, y) \\ & & \circ\!-(\mathsf{empty}(x+2, y+2) \multimap \mathsf{empty}(x+1, y+1) \multimap \mathsf{peg}(x, y) \multimap \mathsf{jump}(n)) \end{array}$$

Finally we add a clause to succeed if we can make n moves given the query $\mathsf{jump}(n)$.

$$done \quad : \quad \mathsf{jump}(0) \circ\!- \top$$

Note the use of $\top$ in order to consume the state at the end of the computation.

The runnable implementation of this program in the concrete syntax of Lolli can be found in the Lolli distribution in the directory `examples/solitaire/`. In order to circumvent the problem that Lolli does not show proofs, but only instantiations for the free variables in the query, we can add another argument to the jump predicate to construct a sequence of moves as it executes.

5.5 Logical Compilation

In this section we examine means to compile program clauses, which may reside either in Γ or Δ. We do not push the ideas very far, but we want to give some intuition on how more efficient and lower level compilation strategies can be developed.

The main question is how to compile procedure call, which, as we have seen, corresponds to applying focusing on the left once the goal is atomic. Instead of working with focusing rules on the left, we would like to translate the program clause to a residual subgoal, still is logically described but which can be interpreted as providing instructions for search (and thereby for computation). More formally, we introduce three new judgments. Note that propositions A

stand for linear hereditary Harrop formulas as before, while G stands for residual formulas whose formal definition we postpone until we have considered what is needed.

$\Gamma; \Delta \Longrightarrow A \upuparrows$	A follows by resolution from $\Gamma; \Delta$
$A \gg P \setminus G$	A immediately entails goal P with residual subgoal G
$\Gamma; \Delta \Longrightarrow G \uparrow$	residual subgoal G has a resolution proof from $\Gamma; \Delta$

The resolution judgment $A \upuparrows$ arises from the uniform proof judgment $A \Uparrow$ by copying all right rules except the choice rules

$$\frac{\Gamma; \Delta; A \Downarrow \Longrightarrow P}{\Gamma; \Delta, A \Longrightarrow P \Uparrow}\,\mathsf{decideL} \qquad \frac{\Gamma, A; \Delta; A \Downarrow \Longrightarrow P}{\Gamma, A; \Delta \Longrightarrow P \Uparrow}\,\mathsf{decideL!}$$

which are replaced by

$$\frac{A \gg P \setminus G \qquad \Gamma; \Delta \Longrightarrow G \uparrow}{\Gamma; \Delta, A \Longrightarrow P \upuparrows}\,\mathsf{decideL}{\gg}$$

$$\frac{A \gg P \setminus G \qquad \Gamma, A; \Delta \Longrightarrow G \uparrow}{\Gamma, A; \Delta \Longrightarrow P \upuparrows}\,\mathsf{decideL!}{\gg}$$

We design the residuation $A \gg P \setminus G$ to be parametric in P, so the translation of A to G does not depend on the precise form of P. In other words, we compile A independently of the form of the call—relaxing this would allow some inlining optimizations. Furthermore, we must be careful that compilation always succeeds and gives a uniquely defined result. That is, for every A and P there exists a unique residual goal G such that $A \gg P \setminus G$. Therefore the natural form of definition is inductive on the structure of A.

On the other hand it turns out our propositions A are not general enough to express residual goals, so we need some additional propositions. We will see for each connective what we need and collect them at the end. Note that if $A \gg P \setminus G$ then a uniform proof with focus formula $A \Downarrow$ and goal P should correspond to a proof of the residual goal $G \uparrow$.

Atomic Propositions. If our focus formula is atomic, then we have to residuate an equality check. Operationally, this will be translated into a call to unification.

$$\frac{}{P' \gg P \setminus P' \doteq P}$$

The right rule for this new connective is clear: it just establishes the equality.

$$\frac{}{\Gamma; \cdot \Longrightarrow P \doteq P \uparrow}\doteq\mathrm{R}$$

For the left rule (which is not needed here), see Exercise 5.3.

Linear Implication. If the focus formula is $A_1 \multimap A_2$, then A_1 is must be solved as a subgoal. In addition, the residuation of A_2 yields another subgoal G. These must be joined with simultaneous conjunction in order to reflect the context split in the $\multimap$L rule.

$$\frac{A_2 \gg P \setminus G}{A_1 \multimap A_2 \gg P \setminus G \otimes A_1}$$

The order is determined by our general convention on conjunctive non-determinism: we first solve G and then A, so G is written on the left. The corresponding right rule for $\otimes$ is familiar, but used here in a slight different form, so we restate it explicitly.

$$\frac{\Gamma; \Delta_1 \Longrightarrow G \uparrow \qquad \Gamma; \Delta_2 \Longrightarrow A \uparrow\uparrow}{\Gamma; \Delta_1, \Delta_2 \Longrightarrow G \otimes A \uparrow} \otimes\mathrm{R}$$

Of course, we use resource management as shown before to postpone the splitting of $\Delta = (\Delta_1, \Delta_2)$ when using this rule during logic program execution.

Alternative Conjunction. If the focus formula is an alternative conjunction $A_1 \& A_2$, then we must choose either to use the $\&\mathrm{L}_1$ or $\&\mathrm{L}_2$ rule. This choice is residuated into an external choice, for which we have the two corresponding rules $\oplus\mathrm{R}_1$ and $\oplus\mathrm{R}_2$.

$$\frac{A_1 \gg P \setminus G_1 \qquad A_2 \gg P \setminus G_2}{A_1 \& A_2 \gg P \setminus G_1 \oplus G_2}$$

Again, we repeat the two right rules for $\oplus$.

$$\frac{\Gamma; \Delta \Longrightarrow G_1 \uparrow}{\Gamma; \Delta \Longrightarrow G_1 \oplus G_2 \uparrow} \oplus\mathrm{R}_1 \qquad \frac{\Gamma; \Delta \Longrightarrow G_2 \uparrow}{\Gamma; \Delta \Longrightarrow G_1 \oplus G_2 \uparrow} \oplus\mathrm{R}_2$$

Additive Truth. If the focus formula is $\top$ then there is no left rule that can be applied and focusing fails. Correspondingly, there is no right rule for $\mathbf{0}$, so $\top$ residuates to $\mathbf{0}$.

$$\frac{}{\top \gg P \setminus \mathbf{0}}$$

As usual, this can be seen as a zero-ary alternative conjunction. And there is no right rule for $\mathbf{0} \uparrow$.

Unrestricted Implication. If the focus formula is an unrestricted implication $A_1 \supset A_2$ we have to solve A_1 as a subgoal, but with access to any of the linear resources.

$$\frac{A_2 \gg P \setminus G}{A_1 \supset A_2 \gg P \setminus G \otimes! A_1}$$

Here we should think of $\otimes!$ as a single binary connective, a point overlooked in [CHP00]. The reason is that it should be statically obvious when solving G

in a proposition $G \otimes ! A_1$ that A_1 cannot consume any resources remaining from G. The new connective is characterized by the following right rule.

$$\frac{\Gamma; \Delta \Longrightarrow G \uparrow \qquad \Gamma; \cdot \Longrightarrow A \uparrow\uparrow}{\Gamma; \Delta \Longrightarrow G \otimes ! A \uparrow} \otimes ! \mathrm{R}$$

For the left rule (which is not needed here) see Exercise 5.2.

Universal Quantification. The universal quantifier is simply residuated into a corresponding existential quantifier.

$$\frac{[a/x]A \gg P \setminus [a/x]G}{\forall x.\ A \gg P \setminus \exists x.\ A}\,a$$

where a must be a new parameter. The new version of the right rule is just as before with a new judgment

$$\frac{\Gamma; \Delta \Longrightarrow [t/x]G \uparrow}{\Gamma; \Delta \Longrightarrow \exists x.\ G \uparrow} \exists \mathrm{R}$$

Thus we needed the following language of *residual goals* G where A ranges over linear hereditary Harrop formulas.

$$G \quad ::= \quad P' \doteq P \mid G \otimes A \mid G_1 \oplus G_2 \mid \mathbf{0} \mid G \otimes ! A \mid \exists x.\ G$$

We call the system of uniform proofs where we replace the choice rules $\mathsf{decideL}$ and $\mathsf{decideL!}$ with $\mathsf{decideL}\gg$ and $\mathsf{decideL!}\gg$ the system of *resolution*. Now the have the following theorem asserting the correctness of resolution.

Theorem 5.3 (Correctness of Resolution)
$\Gamma; \Delta \Longrightarrow A \Uparrow$ *if and only if* $\Gamma; \Delta \Longrightarrow A \uparrow\uparrow$.

Proof: See Exercise 5.1 □

To see how subgoal residuation is related to compilation, consider a simple concrete program that implements a test if a natural number in unary presentation is even.

$$\mathsf{even}(0) \& \forall x.\ \mathsf{even}(x) \multimap \mathsf{even}(\mathsf{s}(\mathsf{s}(x)))$$

Let us call the proposition E. For illustration purposes, we have combined the unrestricted clauses defining even into one using alternative conjunction, because *unrestricted* assumptions A, B are equivalent to a single unrestricted assumption $A \& B$. If we have generic goal $\mathsf{even}(t)$ for some (unknown) term t, we can calculate

$$E \gg \mathsf{even}(t) \setminus \mathsf{even}(0) \doteq \mathsf{even}(t) \oplus \exists x.\ \mathsf{even}(\mathsf{s}(\mathsf{s}(x))) \doteq \mathsf{even}(t) \otimes \mathsf{even}(x)$$

Now we can read the new connectives in the residual goal as search instructions from left to right:

1. Unify ($\doteq$) the proposition $\mathsf{even}(0)$ with the goal $\mathsf{even}(t)$.
2. If this fails ($\oplus$) then create a new existential variable X ($\exists x$).
3. Then unify ($\doteq$) the proposition $\mathsf{even}(\mathsf{s}(\mathsf{s}(X)))$ with the goal $\mathsf{even}(t)$.
4. If this succeeds ($\otimes$) continue with a recursive call to even with X.

Given the logical reading, we can reason about optimizations using the laws of linear logic. For example, $\mathsf{even}(0) \doteq \mathsf{even}(t)$ succeeds exactly if $0 \doteq t$. In the second line we can unfold the unification further, since one of the terms to be unified is known to be $\mathsf{s}(\mathsf{s}(x))$. This kinds of optimization naturally lead to a higher-level version of the Warren Abstract Machine (WAM) [AK91] which is the basis for almost all modern Prolog compilers.

5.6 Exercises

Exercise 5.1 Carefully prove the correctness of resolution (Theorem 5.3). Explicitly state and prove any generalized induction hypotheses or lemmas you might need.

Exercise 5.2 We consider the new connective $\otimes!$ from subgoal residuation as a connective in full intuitionistic linear logic $A \otimes! B$. The right rule in the sequent calculus is determined by the right rule given in Section 5.5.

$$\frac{\Gamma; \Delta \Longrightarrow A \qquad \Gamma; \cdot \Longrightarrow B}{\Gamma; \Delta \Longrightarrow A \otimes! B} \otimes! \mathrm{R}$$

This corresponds directly to the introduction rule in natural deduction.

1. Show the elimination rule(s) in the natural deduction.
2. Show these rule(s) to be locally sound and complete.
3. Show the corresponding left rule(s) in the sequent calculus.
4. Show the new essential cases in the proof of admissibility of cut.

This demonstrates that $\otimes!$ can be seen as a first-class connective of linear logic, even if $A \otimes! B$ may also be considered to as a shorthand for $A \otimes (!B)$.

Exercise 5.3 We consider the new connective $\doteq$ from subgoal residuation as a connective in full intuitionistic linear logic $A \doteq B$. The right rule in the sequent calculus is determined by the right rule given in Section 5.5:

$$\frac{}{\Gamma; \cdot \Longrightarrow A \doteq A} \doteq \mathrm{R}$$

As usual, this corresponds directly to the introduction rule in natural deduction.

1. Show the elimination rule(s) in the natural deduction.
2. Show these rule(s) to be locally sound and complete.
3. Show the corresponding left rule(s) in the sequent calculus.
4. Show how to modify the proof of admissibility of cut to account for the new connective.

Exercise 5.4 Extend the translation to A-normal form, as used in the encoding of the solitaire game, to arbitrary propositions in linear logic. It should have the property that each proposition is translated into a corresponding proposition using only the additional predicate p. State explicitly which laws you need that are similar to uncurrying to eliminate gratuitous uses of propositions that are left asynchronous.

Exercise 5.5 Implement the Towers of Hanoi puzzle with three pegs and several disks in Lolli.

Exercise 5.6 Define a translation that maps a regular expressions r to a predicates p and some unrestricted linear hereditary Harrop formulas. This translation, to be described on paper, should have the property that a word w is in the language of r if and only if $p(w)$ can be proven by the logic programming interpreter underlying Lolli. Implement words as list of single character constants and make sure to pay attention to termination issues.

Apply your translation to $r = a(b + c)^* a$ and execute the resulting program on several successful and failing queries.

Chapter 6

Linear λ-Calculus

In philosophy we distinguish between the notion of *analytic* and *synthetic* judgment [ML94], a terminology which goes back to Kant. Briefly, an analytic judgment can be seen to be evident by virtue of the terms contained in it. A synthetic judgment, on the other hand, requires to go beyond the judgment itself to find evidence for it. The various judgments of truth we have considered such as $\Gamma; \Delta \vdash A$ *true* are *synthetic* because we need the derivation as evidence external to the judgment. We can contrast this with the judgment A *prop* which is *analytic*: an analysis of A itself is sufficient to see if it is a proposition.

It is important to recognize analytic judgments because we do not need to communicate external evidence for them if we want to convince someone of it. The judgment itself carries the evidence. A standard way to convert a synthetic to a corresponding analytic judgment is to enrich it with a term that carries enough information to reconstruct its deduction. We refer to such objects as *proof terms* when they are used to establish the truth of a proposition. There still is a fair amount of latitude in designing proof terms, but with a few additional requirements discussed below they are essentially determined by the structure of the inference rules.

From our intuitionistic point of view it should not be surprising that such proof terms describe constructions. For example, a proof of $A \multimap B$ describes a construction for achieving the goal B given resource A. This can be seen as a plan or a program. In (unrestricted) intuitionistic logic, the corresponding observation that proofs are related to *functional* programs via the *Curry-Howard isomorphism* has been made by [CF58] and [How80]. Howard observed that there is a bijective correspondence between proofs in intuitionistic propositional natural deduction and simply-typed λ-terms. A related observation on proof in combinatory logic had been made previously by Curry [CF58].

A generalization of this observation to include quantifiers gives rise to the rich field of type theory, which we will analyze in Chapter 7. Here we study the basic correspondence, extended to the case of linear logic.

A linear λ-calculus of proof terms will be useful for us in various circumstances. First of all, it gives a compact and faithful representation of proofs as terms. Proof checking is reduced to type-checking in a λ-calculus. For example, if we do not trust the implementation of our theorem prover, we can instrument it to generate proof terms which can be verified independently. In this scenario we are just exploiting that validity of proof terms is an analytic judgment. Secondly, the terms in the λ-calculus provide the core of a functional language with an expressive type system, in which statements such as "*this function will use its argument exactly once*" can be formally expressed and checked. It turns out that such properties can exploited for introducing imperative (state-related) concepts into functional programming [Hof00b], structural complexity theory [Hof00a, AS01], or analysis of memory allocation [WW01] Thirdly, linear λ-terms can serve as an expressive representation language within a *logical framework*, a general meta-language for the formalization of deductive systems.

6.1 Proof Terms

We now assign proof terms to the system of linear natural deduction. Our main criterion for the design of the proof term language is that the proof terms should reflect the structure of the deduction as closely as possible. Moreover, we would like every valid proof term to uniquely determine a natural deduction. Because of weakening for unrestricted hypotheses and the presence of $\top$, this strong property will fail, but a slightly weaker and, from the practical point of view, sufficient property holds. Under the Curry-Howard isomorphism, a proposition corresponds to a type in the proof term calculus. We will there call a proof term *well-typed* if it represents a deduction.

The proof term assignment is defined via the judgment $\Gamma; \Delta \vdash M : A$, where each formula in Γ and Δ is labelled. We also use $M \longrightarrow_\beta M'$ for the local reduction and $M : A \longrightarrow_\eta M'$ for the local expansion, both expressed on proof terms. The type on the left-hand side of the expansion reminds is a reminder that this rule only applies to term of the given type (contexts are elided here).

Hypotheses. We use the label of the hypotheses as the name for a variable in the proof terms. There are no reductions or expansions specific to variables, although variables of non-atomic type may be expanded by the later rules.

$$\frac{}{\Gamma; u{:}A \vdash u : A}\,u \qquad \frac{}{(\Gamma, v{:}A); \cdot \vdash v : A}\,u$$

Recall that we take exchange for granted so that in the rule for unrestricted hypotheses, $v{:}A$ could occur anywhere.

Multiplicative Connectives. Linear implication corresponds to a *linear function types* with corresponding linear abstraction and application. We distinguish them from unrestricted abstraction and application by a "hat". In certain circumstances, this may be unnecessary, but here we want to reflect the proof structure as directly as possible.

$$\frac{\Gamma; (\Delta, u{:}A) \vdash M : B}{\Gamma; \Delta \vdash \hat{\lambda} u{:}A.\ M : A \multimap B} \multimap \mathrm{I}$$

$$\frac{\Gamma; \Delta \vdash M : A \multimap B \qquad \Gamma; \Delta' \vdash N : A}{\Gamma; (\Delta, \Delta') \vdash M \,\hat{}\, N : B} \multimap \mathrm{E}$$

$$\begin{array}{rcl} (\hat{\lambda} w{:}A.\ M) \,\hat{}\, N & \longrightarrow_\beta & [N/w]M \\ M : A \multimap B & \longrightarrow_\eta & \hat{\lambda} w{:}A.\ M \,\hat{}\, w \end{array}$$

In the rules for the simultaneous conjunction, the proof term for the elimination inference is a let form which deconstructs a pair, naming the components. The linearity of the two new hypotheses means that the variables must both be used in M.

$$\frac{\Gamma; \Delta_1 \vdash M : A \qquad \Gamma; \Delta_2 \vdash N : B}{\Gamma; (\Delta_1, \Delta_2) \vdash M \otimes N : A \otimes B} \otimes \mathrm{I}$$

$$\frac{\Gamma; \Delta \vdash M : A \otimes B \quad \Gamma; (\Delta', u{:}A, w{:}B) \vdash N : C}{\Gamma; (\Delta, \Delta') \vdash \mathsf{let}\, u \otimes w = M \,\mathsf{in}\, N : C} \otimes \mathrm{E}$$

The reduction and expansion mirror the local reduction and expansion for deduction as the level of proof terms. We do not reiterate them here, but simply give the proof term reduction.

$$\begin{array}{rcl} \mathsf{let}\, w_1 \otimes w_2 = M_1 \otimes M_2 \,\mathsf{in}\, N & \longrightarrow_\beta & [M_1/w_1, M_2/w_2]N \\ M : A \otimes B & \longrightarrow_\eta & \mathsf{let}\, w_1 \otimes w_2 = M \,\mathsf{in}\, w_1 \otimes w_2 \end{array}$$

The unit type allows us to consume linear hypotheses without introducing new linear ones.

$$\frac{}{\Gamma; \cdot \vdash \star : \mathbf{1}} \mathbf{1}\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash M : \mathbf{1} \qquad \Gamma; \Delta' \vdash N : C}{\Gamma; (\Delta, \Delta') \vdash \mathsf{let}\, \star = M \,\mathsf{in}\, N : C} \mathbf{1}\mathrm{E}$$

$$\begin{array}{rcl} \mathsf{let}\, \star = M \,\mathsf{in}\, N & \longrightarrow_\beta & N \\ M : \star & \longrightarrow_\eta & \mathsf{let}\, \star = M \,\mathsf{in}\, \star \end{array}$$

Additive Connectives. As we have seen from the embedding of intuitionistic in linear logic, the simultaneous conjunction represents products from the simply-typed λ-calculus.

$$\frac{\Gamma; \Delta \vdash M : A \qquad \Gamma; \Delta \vdash N : B}{\Gamma; \Delta \vdash \langle M, N \rangle : A \& B} \& \mathrm{I}$$

$$\frac{\Gamma; \Delta \vdash M : A \& B}{\Gamma; \Delta \vdash \mathsf{fst}\, M : A} \& \mathrm{E_L} \qquad \frac{\Gamma; \Delta \vdash M : A \& B}{\Gamma; \Delta \vdash \mathsf{snd}\, M : B} \& \mathrm{E_R}$$

The local reduction are also the familiar ones.

$$\begin{array}{rcl}
\mathsf{fst}\,\langle M_1, M_2\rangle & \longrightarrow_\beta & M_1 \\
\mathsf{snd}\,\langle M_1, M_2\rangle & \longrightarrow_\beta & M_2 \\[4pt]
M : A \& B & \longrightarrow_\eta & \langle \mathsf{fst}\, M, \mathsf{snd}\, M\rangle
\end{array}$$

The additive unit corresponds to a unit type with no operations on it.

$$\frac{}{\Gamma; \Delta \vdash \langle\,\rangle : \top}\,\top\mathrm{I} \qquad\qquad \textit{No } \top \textit{ elimination}$$

The additive unit has no elimination and therefore no reduction. However, it still admits an expansion, which witnesses the local completeness of the rules.

$$M : \top \quad \longrightarrow_\eta \quad \langle\,\rangle$$

The disjunction (or *disjoint sum* when viewed as a type) uses injection and case as constructor and destructor forms, respectively. We annotated the injections with a type to preserve the property that any well-typed term has a unique type.

$$\frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \mathsf{inl}^B\, M : A \oplus B}\,\oplus\mathrm{I_L} \qquad \frac{\Gamma; \Delta \vdash M : B}{\Gamma; \Delta \vdash \mathsf{inr}^A\, M : A \oplus B}\,\oplus\mathrm{I_R}$$

$$\frac{\Gamma; \Delta \vdash M : A \oplus B \quad \Gamma; (\Delta', w_1{:}A) \vdash N_1 : C \quad \Gamma; (\Delta', w_2{:}B) \vdash N_2 : C}{\Gamma; (\Delta', \Delta) \vdash \mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, w_1 \Rightarrow N_1 \mid \mathsf{inr}\, w_2 \Rightarrow N_2 : C}\,\oplus\mathrm{E}^{w_1, w_2}$$

The reductions are just like the ones for disjoint sums in the simply-typed λ-caclulus.

$$\begin{array}{rcl}
\mathsf{case}\, \mathsf{inl}^B\, M\, \mathsf{of}\, \mathsf{inl}\, w_1 \Rightarrow N_1 \mid \mathsf{inr}\, w_2 \Rightarrow N_2 & \longrightarrow_\beta & [M/w_1]N_1 \\
\mathsf{case}\, \mathsf{inr}^A\, M\, \mathsf{of}\, \mathsf{inl}\, w_1 \Rightarrow N_1 \mid \mathsf{inr}\, w_2 \Rightarrow N_2 & \longrightarrow_\beta & [M/w_2]N_2
\end{array}$$

$$M : A \oplus B \quad \longrightarrow_\eta \quad \mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, w_1 \Rightarrow \mathsf{inl}^B\, w_1 \mid \mathsf{inr}\, w_2 \Rightarrow \mathsf{inr}^A\, w_2$$

For the additive falsehood, there is no introduction rule. It corresponds to a *void type* without any values. Consequently, there is no reduction. Once again we annotate the abort constructor in order to guarantee uniqueness of types.

$$\textit{No } \mathbf{0} \textit{ introduction} \qquad\qquad \frac{\Gamma; \Delta \vdash M : \mathbf{0}}{\Gamma; (\Delta, \Delta') \vdash \mathsf{abort}^C\, M : C}\,\mathbf{0}\mathrm{E}$$

$$M : \mathbf{0} \quad \longrightarrow_\eta \quad \mathsf{abort}^{\mathbf{0}}\, M$$

Exponentials. Unrestricted implication corresponds to the usual *function type* from the simply-typed λ-calculus. For consistency, we will still write $A \supset B$ instead of $A \rightarrow B$, which is more common in λ-calculus. Note that the argument of an unrestricted application may not mention any linear variables.

$$\frac{(\Gamma, v{:}A); \Delta \vdash M : B}{\Gamma; \Delta \vdash \lambda v{:}A.\ M : A \supset B} \supset\mathrm{I}$$

$$\frac{\Gamma; \Delta \vdash M : A \supset B \qquad \Gamma; \cdot \vdash N : A}{\Gamma; \Delta \vdash M\, N : B} \supset\mathrm{E}$$

The reduction and expansion are the origin of the β and η rules names due to Church [Chu41].

$$\begin{array}{rcl} (\lambda v{:}A.\ M)\, N & \longrightarrow_\beta & [N/v]M \\ M : A \supset B & \longrightarrow_\eta & \lambda v{:}A.\ M\, v \end{array}$$

The rules for the *of course* operator allow us to name term of type $!A$ and use it freely in further computation.

$$\frac{\Gamma; \cdot \vdash M : A}{\Gamma; \cdot \vdash\, !M : !A} !\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash M : !A \qquad (\Gamma, v{:}A); \Delta' \vdash N : C}{\Gamma; (\Delta', \Delta) \vdash \mathsf{let}\ !v = M\ \mathsf{in}\ N : C} !\mathrm{E}$$

$$\begin{array}{rcl} \mathsf{let}\ !v = !M\ \mathsf{in}\ N & \longrightarrow_\beta & [M/v]N \\ M : !A & \longrightarrow_\eta & \mathsf{let}\ !v = M\ \mathsf{in}\ !v \end{array}$$

Below is a summary of the linear λ-calculus with the β-reduction and η-expansion rules.

$$\begin{array}{rcll} M & ::= & u & \textit{Linear Variables} \\ & & \mid \hat{\lambda} u{:}A.\ M \mid M_1 \hat{\ } M_2 & A \multimap B \\ & & \mid M_1 \otimes M_2 \mid \mathsf{let}\ u_1 \otimes u_2 = M\ \mathsf{in}\ M' & A \otimes B \\ & & \mid \star \mid \mathsf{let}\ \star = M\ \mathsf{in}\ M' & \mathbf{1} \\ & & \mid \langle M_1, M_2 \rangle \mid \mathsf{fst}\ M_1 \mid \mathsf{snd}\ M_2 & A \& B \\ & & \mid \langle\,\rangle & \top \\ & & \mid \mathsf{inl}^B\ M \mid \mathsf{inr}^A\ M & A \oplus B \\ & & \quad \mid (\mathsf{case} M\ \mathsf{of}\ \mathsf{inl}\ u_1 \Rightarrow M_1 \mid \mathsf{inr}\ u_2 \Rightarrow M_2) & \\ & & \mid \mathsf{abort}^C\ M & \mathbf{0} \\ & & \mid v & \textit{Unrestricted Variables} \\ & & \mid \lambda v{:}A.\ M \mid M_1\, M_2 & A \supset B \\ & & \mid !M \mid \mathsf{let}\ v = M\ \mathsf{in}\ M' & !A \end{array}$$

Below is a summary of the β-reduction rules, which correspond to local reductions of natural deductions.

$$
\begin{array}{rcll}
(\hat{\lambda}u{:}A.\ M)\,\hat{}\,N & \longrightarrow_\beta & [N/u]M & A \multimap B \\
\mathsf{let}\ u_1 \otimes u_2 = M_1 \otimes M_2\ \mathsf{in}\ N & \longrightarrow_\beta & [M_1/u_1, M_2/u_2]N & A \otimes B \\
\mathsf{let}\ \star = M\ \mathsf{in}\ N & \longrightarrow_\beta & N & \mathbf{1} \\
\mathsf{fst}\ \langle M_1, M_2\rangle & \longrightarrow_\beta & M_1 & A \& B \\
\mathsf{snd}\ \langle M_1, M_2\rangle & \longrightarrow_\beta & M_2 & \\
\textit{No } \top \textit{ reduction} & & & \\
\mathsf{case}\ \mathsf{inl}^B\ M\ \mathsf{of}\ \mathsf{inl}\ u_1 \Rightarrow N_1 \mid \mathsf{inr}\ u_2 \Rightarrow N_2 & \longrightarrow_\beta & [M/u_1]N_1 & A \oplus B \\
\mathsf{case}\ \mathsf{inr}^A\ M\ \mathsf{of}\ \mathsf{inl}\ u_1 \Rightarrow N_1 \mid \mathsf{inr}\ u_2 \Rightarrow N_2 & \longrightarrow_\beta & [M/u_1]N_2 & \\
\textit{No } \mathbf{0} \textit{ reduction} & & & \\
(\lambda v{:}A.\ M)\ N & \longrightarrow_\beta & [N/v]M & A \supset B \\
\mathsf{let}\ !v = !M\ \mathsf{in}\ N & \longrightarrow_\beta & [M/v]N & !A
\end{array}
$$

The substitution $[M/u]N$ and $[M/v]N$ assumes that there are no free variables in M which would be captured by a variables binding in N. We nonethless consider it a total function, since the capturing variable can always be renamed to avoid a conflict (see Exercise 6.3).

Next is a summary of the η-expansion rules, which correspond to local expansions of natural deductions.

$$
\begin{array}{rcl}
M : A \multimap B & \longrightarrow_\eta & \hat{\lambda}u{:}A.\ M\,\hat{}\,u \\
M : A \otimes B & \longrightarrow_\eta & \mathsf{let}\ u_1 \otimes u_2 = M\ \mathsf{in}\ u_1 \otimes u_2 \\
M : \star & \longrightarrow_\eta & \mathsf{let}\ \star = M\ \mathsf{in}\ \star \\
M : A \& B & \longrightarrow_\eta & \langle \mathsf{fst}\ M, \mathsf{snd}\ M\rangle \\
M : \top & \longrightarrow_\eta & \langle\,\rangle \\
M : A \oplus B & \longrightarrow_\eta & \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ u_1 \Rightarrow \mathsf{inl}^B\ u_1 \mid \mathsf{inr}\ u_2 \Rightarrow \mathsf{inr}^A\ u_2 \\
M : \mathbf{0} & \longrightarrow_\eta & \mathsf{abort}^{\mathbf{0}}\ M \\
M : A \supset B & \longrightarrow_\eta & \lambda v{:}A.\ M\ v \\
M : !A & \longrightarrow_\eta & \mathsf{let}\ !v = M\ \mathsf{in}\ !v
\end{array}
$$

Note that there is an implicit assumption that the variables w and u in the cases for $A \multimap B$ and $A \supset B$ do not already occur in M: they are chosen to be new.

If $\mathcal{P}$ is a derivation of $\Gamma; \Delta \vdash M : A$ then we write erase($\mathcal{P}$) for the corresponding derivation of $\Gamma; \Delta \vdash A$ *true* where the proof term M has been erased from every judgment.

We have the following fundamental properties. Uniqueness, where claimed, holds only up to renaming of bound variables.

Theorem 6.1 (Properties of Proof Terms)

1. *If $\mathcal{P}$ is a derivation of $\Gamma; \Delta \vdash M : A$ then* erase($\mathcal{P}$) *is a derivation of $\Gamma; \Delta \vdash A$.*

2. *If $\mathcal{D}$ is a derivation of $\Gamma; \Delta \vdash A$ then there is a unique M and derivation $\mathcal{P}$ of $\Gamma; \Delta \vdash M : A$ such that* erase($\mathcal{P}$) $= \mathcal{D}$.

Proof: By straightforward inductions over the given derivations. □

Types are also unique for well-typed terms (see Exercise 6.1). Uniqueness of derivations fails, that is, a proof term does not uniquely determine its derivation, even under identical contexts. A simple counterexample is provided by the following two derivations (with the empty unrestricted context elided).

$$\dfrac{\dfrac{}{w{:}\top \vdash \langle\,\rangle : \top}\top I \qquad \dfrac{}{\cdot \vdash \langle\,\rangle : \top}\top I}{w{:}\top \vdash \langle\,\rangle \otimes \langle\,\rangle : \top \otimes \top}\otimes I \qquad \dfrac{\dfrac{}{\cdot \vdash \langle\,\rangle : \top}\top I \qquad \dfrac{}{w{:}\top \vdash \langle\,\rangle : \top}\top I}{w{:}\top \vdash \langle\,\rangle \otimes \langle\,\rangle : \top \otimes \top}\otimes I$$

It can be shown that linear hypotheses which are absorbed by $\top I$ are the only source of only ambiguity in the derivation. A similar ambiguity already exists in the sense that any proof term remains valid under weakening in the unrestricted context: whenever $\Gamma; \Delta \vdash M : A$ then $(\Gamma, \Gamma'); \Delta \vdash M : A$. So this phenomenon is not new to the linear λ-calculus, and is in fact a useful identification of derivations which differ in "irrelevant" details, that is, unused or absorbed hypotheses.

The substitution principles on natural deductions can be expressed on proof terms. This is because the translations from natural deductions to proof terms and *vice versa* are *compositional*: uses of a hypothesis labelled u in natural deduction corresponds to an occurrence of a variable u in the proof term.

Lemma 6.2 (Substitution on Proof Terms)

1. *If* $\Gamma; (\Delta, u{:}A) \vdash N{:}C$ *and* $\Gamma; \Delta' \vdash M : A$, *then* $\Gamma; (\Delta, \Delta') \vdash [M/u]N : C$.
2. *If* $(\Gamma, v{:}A); \Delta \vdash N{:}C$ *and* $\Gamma; \cdot \vdash M : A$, *then* $\Gamma; \Delta \vdash [M/v]N : C$.

Proof: By induction on the structure of the first given derivation, using the property of exchange. □

We also have the property of weakening for unrestricted hypotheses. The substitution properties are the critical ingredient for the important *subject reduction* properties, which guarantee that the result of β-reducing a well-typed term will again be well-typed. The expansion rules also preserve types when invoked properly.

Theorem 6.3 (Subject Reduction and Expansion)

1. *If* $\Gamma; \Delta \vdash M : A$ *and* $M \longrightarrow_\beta M'$ *then* $\Gamma; \Delta \vdash M' : A$.
2. *If* $\Gamma; \Delta \vdash M : A$ *and* $M : A \longrightarrow_\eta M'$ *then* $\Gamma; \Delta \vdash M' : A$.

Proof: For subject reduction we examine each possible reduction rule, applying inversion to obtain the shape of the typing derivation. From this we either directly construct the typing derivation of M' or we appeal to the substitution lemma.

For subject expansion we directly construct the typing derivation for M' from the typing derivation of M. □

Note that the opposite of subject reduction does not hold: there are well-typed terms M' such that $M \longrightarrow_\beta M'$ and M is not well-typed (see Exercise 6.4).

6.2 Linear Type Checking

The typing rules for the linear λ-calculus are *syntax-directed* in that the principal term constructor determines the typing rule which must be used. Nonetheless, the typing rules are not immediately suitable for an efficient type-checking algorithm since we would have to guess how the linear hypotheses are to be split between the hypothesis in a number of rules.

The occurrence constraints introduced in Section **??** would be sufficient to avoid this choice, but they are rather complex, jeopardizing our goal of designing a simple procedure which is easy to trust. Fortunately, we have significantly more information here, since the proof term is given to us. This determines the amount of work we have to do in each branch of a derivation, and we can resolve the don't-care non-determinism directly.

Instead of guessing a split of the linear hypotheses between two premisses of a rule, we pass all linear variables to the first premiss. Checking the corresponding subterm will consume some of these variables, and we pass the remaining ones one to check the second subterms. This idea requires a judgment

$$\Gamma; \Delta_I \setminus \Delta_O \vdash M : A$$

where Δ_I represents the available linear hypotheses and $\Delta_O \subseteq \Delta_I$ the linear hypotheses not used in M. For example, the rules for the simultaneous conjunction and unit would be

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash M : A \qquad \Gamma; \Delta' \setminus \Delta_O \vdash N : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash M \otimes N : A \otimes B} \otimes\mathrm{I}$$

$$\frac{}{\Gamma; \Delta_I \setminus \Delta_I \vdash \star : A} \mathbf{1}\mathrm{I}.$$

Unfortunately, this idea breaks down when we encounter the additive unit (and only then!). Since we do not know which of the linear hypotheses might be used in a different branch of the derivation, it would have to read

$$\frac{\Delta_I \supseteq \Delta_O}{\Gamma; \Delta_I \setminus \Delta_O \vdash \langle\,\rangle : \top} \top\mathrm{I}$$

which introduces undesirable non-determinism if we were to guess which subset of Δ_I to return. In order to cirumvent this problem we return all of Δ_I, but flag it to indicate that it may not be exact, but that some of these linear hypotheses may be absorbed if necessary. In other words, in the judgment

$$\Gamma; \Delta_I \setminus \Delta_O \vdash_1 M : A$$

any of the remaining hypotheses in Δ_O need not be consumed in the other branches of the typing derivation. On the other hand, the judgment

$$\Gamma; \Delta_I \setminus \Delta_O \vdash_0 M : A$$

indicates the M uses exactly the variables in $\Delta_I - \Delta_O$.

When we think of the judgment $\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A$ as describing an algorithm, we think of Γ, Δ_I and M as given, and Δ_O and the slack indicator i as part of the result of the computation. The type A may or may not be given—in one case it is synthesized, in the other case checked. This refines our view as computation being described as the bottom-up construction of a derivation to include parts of the judgment in different roles (as input, output, or bidirectional components). In logic programming, which is based on the notion of computation-as-proof-search, these roles of the syntactic constituents of a judgment are called *modes*. When writing a deductive system to describe an algorithm, we have to be careful to respect the modes. We discuss this further when we come to the individual rules.

Hypotheses. The two variable rules leave no slack, since besides the hypothesis itself, no assumptions are consumed.

$$\frac{}{\Gamma; (\Delta_I, w{:}A) \setminus \Delta_I \vdash_0 w : A}\,w \qquad \frac{}{(\Gamma, u{:}A); \Delta_I \setminus \Delta_I \vdash_0 u : A}\,u$$

Multiplicative Connectives. For linear implication, we must make sure that the hypothesis introduced by $\multimap$I actually was used and is not part of the residual hypothesis Δ_O. If there is slack, we can simply erase it.

$$\frac{\Gamma; (\Delta_I, w{:}A) \setminus \Delta_O \vdash_i M : B \quad \text{where } i = 1 \text{ or } w \text{ not in } \Delta_O}{\Gamma; \Delta_I \setminus (\Delta_O - w{:}A) \vdash_i \hat{\lambda} w{:}A.\, M : A \multimap B}\,\multimap \mathrm{I}^w$$

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash_i M : A \multimap B \qquad \Gamma; \Delta' \setminus \Delta_O \vdash_k N : A}{\Gamma; (\Delta_I \setminus \Delta_O) \vdash_{i \vee k} M\,\hat{}\,N : B}\,\multimap \mathrm{E}$$

Here $i \vee k = 1$ if $i = 1$ or $k = 1$, and $i \vee k = 0$ otherwise. This means we have slack in the result, if either of the two premisses permits slack.

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash_i M : A \qquad \Gamma; \Delta' \setminus \Delta_O \vdash_k N : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_{i \vee k} M \otimes N : A \otimes B}\,\otimes\mathrm{I}$$

$$\frac{\begin{array}{l}\Gamma; \Delta_I \setminus \Delta' \vdash_i M : A \otimes B \\ \Gamma; (\Delta', w_1{:}A, w_2{:}B) \setminus \Delta_O \vdash_k N : C \\ \text{where } k = 1 \text{ or } w_1 \text{ and } w_2 \text{ not in } \Delta_O\end{array}}{\Gamma; \Delta_I \setminus (\Delta_O - w_1{:}A - w_2{:}B) \vdash_{i \vee k} \mathsf{let}\, w_1 \otimes w_2 = M \,\mathsf{in}\, N : C}\,\otimes\mathrm{E}^{w_1,w_2}$$

In the ⊗E rule we stack the premisses on top of each other since they are too long to fit on one line. The unit type permits no slack.

$$\frac{}{\Gamma; \Delta_I \setminus \Delta_I \vdash_0 \star : \mathbf{1}}\ \mathbf{1}\mathrm{I}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash_i M : \mathbf{1} \qquad \Gamma; \Delta' \setminus \Delta_O \vdash_k N : C}{\Gamma; \Delta_I \setminus \Delta_O \vdash_{i \vee k} \mathsf{let}\, \star = M\, \mathsf{in}\, N : C}\ \mathbf{1}\mathrm{E}$$

Additive Connectives. The mechanism of passing and consuming resources was designed to eliminate unwanted non-determinism in the multiplicative connectives. This introduces complications in the additives, since we have to force premisses to consume exactly the same resources. We write out four version of the &I rule.

$$\frac{\Gamma; \Delta_I \setminus \Delta'_O \vdash_0 M : A \quad \Gamma; \Delta_I \setminus \Delta''_O \vdash_0 N : B \quad \Delta'_O = \Delta''_O}{\Gamma; \Delta_I \setminus (\Delta'_O \cap \Delta''_O) \vdash_0 \langle M, N \rangle : A \& B}\ \&\mathrm{I}_{00}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta'_O \vdash_0 M : A \quad \Gamma; \Delta_I \setminus \Delta''_O \vdash_1 N : B \quad \Delta'_O \subseteq \Delta''_O}{\Gamma; \Delta_I \setminus (\Delta'_O \cap \Delta''_O) \vdash_0 \langle M, N \rangle : A \& B}\ \&\mathrm{I}_{10}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta'_O \vdash_1 M : A \quad \Gamma; \Delta_I \setminus \Delta''_O \vdash_0 N : B \quad \Delta'_O \supseteq \Delta''_O}{\Gamma; \Delta_I \setminus (\Delta'_O \cap \Delta''_O) \vdash_0 \langle M, N \rangle : A \& B}\ \&\mathrm{I}_{01}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta'_O \vdash_1 M : A \quad \Gamma; \Delta_I \setminus \Delta''_O \vdash_1 N : B}{\Gamma; \Delta_I \setminus (\Delta'_O \cap \Delta''_O) \vdash_1 \langle M, N \rangle : A \& B}\ \&\mathrm{I}_{11}$$

Note that in $\&\mathrm{I}_{00}$, $\Delta'_O \cap \Delta''_O = \Delta'_O = \Delta''_O$ by the condition in the premiss. Similarly for the other rules. We chose to present the rules in a uniform way despite this redundancy to highlight the similarities. Only if both premisses permit slack do we have slack overall.

$$\frac{\Gamma; \Delta \setminus \Delta_O \vdash_i M : A \& B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \mathsf{fst}\, M : A}\ \&\mathrm{E}_\mathrm{L} \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A \& B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \mathsf{snd}\, M : B}\ \&\mathrm{E}_\mathrm{R}$$

Finally, we come to the reason for the slack indicator.

$$\frac{}{\Gamma; \Delta_I \setminus \Delta_I \vdash_1 \langle\, \rangle : \top}\ \top\mathrm{I} \qquad \textit{No } \top \textit{ elimination}$$

The introduction rules for disjunction are direct.

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \mathsf{inl}^B : A \oplus B}\ \oplus\mathrm{I}_\mathrm{L} \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \mathsf{inr}^A : A \oplus B}\ \oplus\mathrm{I}_\mathrm{R}$$

The elimination rule for disjunction combines resource propagation (as for multiplicatives) introduction of hypothesis, and resource coordination (as for additives) and is therefore somewhat tedious. It is left to Exercise 6.6. The **0**E rule permits slack, no matter whether the derivation of the premiss permits slack.

$$\textit{No } \mathbf{0} \textit{ introduction} \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : \mathbf{0}}{\Gamma; \Delta_I \setminus \Delta_O \vdash_1 \mathsf{abort}^C\, M : C}\,\mathbf{0}\mathrm{E}$$

Exponentials. Here we can enforce the emptiness of the linear context directly.

$$\frac{(\Gamma, u{:}A); \Delta_I \setminus \Delta_O \vdash_i M : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \lambda u{:}A.\ M : A \supset B}\,{\supset}\mathrm{I}^u$$

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A \supset B \qquad \Gamma; \cdot \setminus \Delta^* \vdash_k N : A}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M\, N : B}\,{\supset}\mathrm{E}$$

Here Δ^* will always have to be $\cdot$ (since it must be a subset of $\cdot$) and k is irrelevant. The same is true in the next rule.

$$\frac{\Gamma; \cdot \setminus \Delta^* \vdash_i M : A}{\Gamma; \Delta_I \setminus \Delta_I \vdash_0 {!M} : {!A}}\,!\mathrm{I}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash_i M : {!A} \qquad (\Gamma, u{:}A); \Delta' \setminus \Delta_O \vdash_k N : C}{\Gamma; \Delta_I \setminus \Delta_O \vdash_{i \vee j} \mathsf{let}\, {!u} = M \,\mathsf{in}\, N : C}\,!\mathrm{E}^u$$

The desired soundness and completeness theorem for the algorithmic typing judgment must first be generalized before it can be proved by induction. For this generalization, the mode (input and output) of the constituents of the judgment is a useful guide. For example, in the completness direction (3), we can expect to distinguish cases based on the slack indicator which might be returned when we ask the question if there are Δ_O and i such that $\Gamma; \Delta \setminus \Delta_O \vdash_i M : A$ for the given Γ, Δ, M and A.

Lemma 6.4 (Properties of Algorithmic Type Checking)

1. *If* $\Gamma; \Delta_I \setminus \Delta_O \vdash_0 M : A$ *then* $\Delta_I \supseteq \Delta_O$ *and* $\Gamma; \Delta_I - \Delta_O \vdash M : A$.
2. *If* $\Gamma; \Delta_I \setminus \Delta_O \vdash_1 M : A$ *then* $\Delta_I \supseteq \Delta_O$ *and for any* Δ *such that* $\Delta_I \supseteq \Delta \supseteq \Delta_I - \Delta_O$ *we have* $\Gamma; \Delta \vdash M : A$.
3. *If* $\Gamma; \Delta \vdash M : A$ *then either*
   (a) $\Gamma; (\Delta', \Delta) \setminus \Delta' \vdash_0 M : A$ *for any* Δ', *or*
   (b) $\Gamma; (\Delta', \Delta) \setminus (\Delta', \Delta_O) \vdash_1 M : A$ *for all* Δ' *and some* $\Delta_O \subseteq \Delta$.

Proof: By inductions on the structure of the given derivations.[1] Items (1) and (2) must be proven simultaneously. □

From this lemma, the soundness and completeness of algorithmic type checking follow directly.

Theorem 6.5 (Algorithmic Type Checking)
$\Gamma; \Delta \vdash M : A$ *if and only if either*

1. $\Gamma; \Delta \setminus \cdot \vdash_0 M : A$, *or*
2. $\Gamma; \Delta \setminus \Delta' \vdash_1 M : A$ *for some* Δ'.

Proof: Directly from Lemma 6.4 □

6.3 Pure Linear Functional Programming

The linear λ-calculus developed in the preceding sections can serve as the basis for a programming language. The step from λ-calculus to programming language can be rather complex, depending on how realistic one wants to make the resulting language. The first step is to decide on *observable types* and a language of *values* and then define an *evaluation judgment*. This is the subject of this section. Given the purely logical view we have taken, this language still lacks datatypes and recursion. In order to remedy this, we introduce *recursive types* and *recursive terms* in the next section.

Our operational semantics follows the intuition that we should not evaluate expressions whose value may not be needed for the result. Expressions whose value will definitely be used, can be evaluated eagerly. There is a slight mismatch in that the linear λ-calculus can identifies expressions whose value will be needed *exactly once*. However, we can derive other potential benefits from the stronger restriction at the lower levels of an implementation such as improved garbage collection or update-in-place. These benefits also have their price, and at this time the trade-offs are not clear. For the *strict λ-calculus* which captures the idea of definite use of the value of an expression, see Exercise 6.2.

We organize the functional language strictly along the types, discussing observability, values, and evaluation rules for each. We have two main judgments, M *Value* (M is a value), and $M \hookrightarrow v$ (M evaluates to v). In general we use v for terms which are legal values. For both of these we assume that M is *closed* and *well-typed*, that is, $\cdot; \cdot \vdash M : A$.

Linear Implication. An important difference between a general λ-calculus and a functional language is that the structure of functions in a programming language is *not* observable. Instead, functions are compiled to code. Their behavior can be observed by applying functions to arguments, but their definition cannot be seen. Thus, strictly speaking, it is incorrect to say that functions

[1] [*check*]

are first-class. This holds equally for so-called lazy functional languages such as Haskell and eager functional languages such as ML. Thus, any expression of the form $\hat{\lambda} w{:}A.\ M$ is a possible value.

$$\frac{}{\hat{\lambda} w{:}A.\ M\ \mathit{Value}} \multimap \mathrm{val}$$

Evaluation of a $\hat{\lambda}$-abstraction returns itself immediately.

$$\frac{}{\hat{\lambda} w{:}A.\ M \hookrightarrow \hat{\lambda} w{:}A.\ M} \multimap \mathrm{Iv}$$

Since a linear parameter to a function is definitely used (in fact, used exactly once), we can evaluate the argument without doing unnecessary work and substitute it for the bound variable during the evaluation of an application.

$$\frac{M_1 \hookrightarrow \hat{\lambda} w{:}A_2.\ M_1' \qquad M_2 \hookrightarrow v_2 \qquad [v_2/w]M_1' \hookrightarrow v}{M_1 \,\hat{}\, M_2 \hookrightarrow v} \multimap \mathrm{Ev}$$

Note that after we substitute the value of argument v_2 for the formal parameter w in the function, we have to evaluate the body of the function.

Simultaneous Pairs. The multiplicative conjunction $A \otimes B$ corresponds to the type of pairs where both elements must be used exactly once. Thus we can evaluate the components (they will be used!) and the pairs are observable. The elimination form is evaluated by creating the pair and then deconstructing it.

$$\frac{M_1\ \mathit{Value} \qquad M_2\ \mathit{Value}}{M_1 \otimes M_2\ \mathit{Value}} \otimes\mathrm{val}$$

$$\frac{M_1 \hookrightarrow v_1 \qquad M_2 \hookrightarrow v_2}{M_1 \otimes M_2 \hookrightarrow v_1 \otimes v_2} \otimes\mathrm{Iv} \qquad \frac{M \hookrightarrow v_1 \otimes v_2 \qquad [v_1/w_1, v_2/w_2]N \hookrightarrow v}{\mathsf{let}\ w_1 \otimes w_2 = M\ \mathsf{in}\ N \hookrightarrow v} \otimes\mathrm{Ev}$$

Multiplicative Unit. The multiplicative unit $\mathbf{1}$ is observable and contains exactly one value $\star$. Its elimination rule explicitly evaluates a term and ignores its result (which must be $\star$).

$$\frac{}{\star\ \mathit{Value}} \mathbf{1}\mathrm{val}$$

$$\frac{}{\star \hookrightarrow \star} \mathbf{1}\mathrm{Iv} \qquad \frac{M \hookrightarrow \star \qquad N \hookrightarrow v}{\mathsf{let}\ \star = M\ \mathsf{in}\ N \hookrightarrow v} \mathbf{1}\mathrm{Ev}$$

Alternative Pairs. Alternative pairs of type $A \& B$ are such that we can only use one of the two components. Since we may not be able to predict which one, we should not evaluate the components. Thus pairs $\langle M_1, M_2 \rangle$ are *lazy*, not observable and any pair of this form is a value. When we extract a component, we then have to evaluate the corresponding term to obtain a value.

$$\frac{}{\langle M_1, M_2 \rangle \ \mathit{Value}}\ \&\mathrm{val}$$

$$\frac{}{\langle M_1, M_2 \rangle \hookrightarrow \langle M_1, M_2 \rangle}\ \&\mathrm{Iv}$$

$$\frac{M \hookrightarrow \langle M_1, M_2 \rangle \qquad M_1 \hookrightarrow v_1}{\mathsf{fst}\, M \hookrightarrow v_1}\ \&\mathrm{Ev}_1 \qquad \frac{M \hookrightarrow \langle M_1, M_2 \rangle \qquad M_2 \hookrightarrow v_2}{\mathsf{snd}\, M \hookrightarrow v_2}\ \&\mathrm{Ev}_2$$

Additive Unit. By analogy, the additive unit $\top$ is not observable. Since there is no elimination rule, we can never do anything interesting with a value of this type, except embed it in larger values.

$$\frac{}{\langle\,\rangle \ \mathit{Value}}\ \top\mathrm{val}$$

$$\frac{}{\langle\,\rangle \hookrightarrow \langle\,\rangle}\ \top\mathrm{Iv}$$

This rule does not express the full operational intuition behind $\top$ which "garbage collects" all linear resources. However, we can only fully appreciate this when we define evaluation under environments (see Section **??**).

Disjoint Sum. The values of a disjoint sum type are guaranteed to be used (no matter whether it is of the form $\mathsf{inl}^B\, M$ or $\mathsf{inr}^A\, M$). Thus we can require values to be built up from injections of values, and the structure of sum values is observable. There are two rules for evaluation, depending on whether the subject of a case-expression is a left injection or right injection into the sum type.

$$\frac{M \ \mathit{Value}}{\mathsf{inl}^B\, M \ \mathit{Value}}\ \oplus\mathrm{val}_1 \qquad \frac{M \ \mathit{Value}}{\mathsf{inr}^A\, M \ \mathit{Value}}\ \oplus\mathrm{val}_2$$

$$\frac{M \hookrightarrow v}{\mathsf{inl}^B\, M \hookrightarrow \mathsf{inl}^B\, v}\ \oplus\mathrm{Iv}_1 \qquad \frac{M \hookrightarrow v}{\mathsf{inr}^A\, M \hookrightarrow \mathsf{inr}^A\, v}\ \oplus\mathrm{Iv}_2$$

$$\frac{M \hookrightarrow \mathsf{inl}^B\, v_1 \qquad [v_1/w_1]N_1 \hookrightarrow v}{\mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, w_1 \Rightarrow N_1 \mid \mathsf{inr}\, w_2 \Rightarrow N_2 \hookrightarrow v}\ \oplus\mathrm{Ev}_1$$

$$\frac{M \hookrightarrow \mathsf{inr}^A\, v_2 \qquad [v_2/w_2]N_2 \hookrightarrow v}{\mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, w_1 \Rightarrow N_1 \mid \mathsf{inr}\, w_2 \Rightarrow N_2 \hookrightarrow v}\ \oplus\mathrm{Ev}_2$$

Void Type. The void type $\mathbf{0}$ contains no value. In analogy with the disjoint sum type it is observable, although this is not helpful in practice. There are no evaluation rules for this type: since there are no introduction rules there are no constructor rules, and the elimination rule distinguishes between zero possible cases (in other words, is impossible). We called this $\mathsf{abort}^A\, M$, since it may be viewed as a global program abort.

Unrestricted Function Type. The unrestricted function type $A \supset B$ (also written as $A \rightarrow B$ in accordance with the usual practice in functional programming) may or may not use its argument. Therefore, the argument is not evaluated, but simply substituted for the bound variable. This is referred to as a *call-by-name* semantics. It is usually implemented by *lazy evaluation*, which means that first time the argument is evaluated, this value is memoized to avoid re-evaluation. This is not represented at this level of semantic description. Values of functional type are not observable, as in the linear case.

$$\frac{}{\lambda u{:}A.\ M\ \ \mathit{Value}} \rightarrow \mathrm{Ival}$$

$$\frac{}{\lambda u{:}A.\ M \hookrightarrow \lambda u{:}A.\ M} \rightarrow \mathrm{Iv}$$

$$\frac{M_1 \hookrightarrow \lambda u{:}A_2.\ M_1' \qquad [M_2/u]M_1' \hookrightarrow v}{M_1\ M_2 \hookrightarrow v} \rightarrow \mathrm{Ev}$$

Modal Type. A linear variable of type $!A$ must be used, but the embedded expression of type A may *not* be used since it is unrestricted. Therefore, terms $!M$ are values and "!" is like a quotation of its argument M, protecting it from evaluation.

$$\frac{}{!M\ \ \mathit{Value}}\,!\mathrm{val}$$

$$\frac{}{!M \hookrightarrow\, !M}\,!\mathrm{Iv} \qquad \frac{M \hookrightarrow\, !M' \qquad [M'/u]N \hookrightarrow v}{\mathsf{let}\, !u = M\, \mathsf{in}\, N \hookrightarrow v}\,!\mathrm{Ev}$$

We abbreviate the value judgment from above in the form of a grammar.

$$\begin{array}{llllll}
\mathit{Values} & v & ::= & \hat{\lambda} w{:}A.\ M & A \multimap B & \text{not observable} \\
 & & \mid & v_1 \otimes v_2 & A_1 \otimes A_2 & \text{observable} \\
 & & \mid & \star & \mathbf{1} & \text{observable} \\
 & & \mid & \langle M_1, M_2 \rangle & A_1 \& A_2 & \text{not observable} \\
 & & \mid & \langle\,\rangle & \top & \text{not observable} \\
 & & \mid & \mathsf{inl}^B\, v \mid \mathsf{inr}^A\, v & A \oplus B & \text{observable} \\
 & & & \textit{No values} & \mathbf{0} & \text{observable} \\
 & & \mid & \lambda u{:}A.\ M & A \rightarrow B & \text{not observable} \\
 & & \mid & !M & !A & \text{not observable}
\end{array}$$

In the absence of datatypes, we cannot write many interesting programs. As a first example we consider the representation of the Booleans with two values, true and false, and a conditional as an elimination construct.

$$
\begin{array}{rcl}
\text{bool} & = & \mathbf{1} \oplus \mathbf{1} \\
\text{true} & = & \mathsf{inl}^{\mathbf{1}}\, \star \\
\text{false} & = & \mathsf{inr}^{\mathbf{1}}\, \star \\
\textbf{if}\, M\, \textbf{then}\, N_1\, \textbf{else}\, N_2 & = & \mathsf{case}\, M \\
& & \quad \mathsf{of}\ \mathsf{inl}^{\mathbf{1}}\, w_1 \Rightarrow \mathsf{let}\, \star = w_1\, \mathsf{in}\, N_1 \\
& & \quad \ \mid \mathsf{inr}^{\mathbf{1}}\, w_2 \Rightarrow \mathsf{let}\, \star = w_2\, \mathsf{in}\, N_2
\end{array}
$$

The elimination of $\star$ in the definition of the conditional is necessary, because a branch $\mathsf{inl}^{\mathbf{1}}\, w_1 \Rightarrow N_1$ would not be well-typed: w_1 is a linear variable not used in its scope. Destructuring a value in several stages is a common idiom and it is helpful for the examples to introduce some syntactic sugar. We allow patterns which nest the elimination forms which appear in a let or case. Not all combination of these are legal, but it is not difficult to describe the legal pattern and match expressions (see Exercise 6.7).

$$
\begin{array}{llcl}
\textit{Patterns} & p & ::= & w \mid p_1 \otimes p_2 \mid \star \mid \mathsf{inl}\, p \mid \mathsf{inr}\, p \mid u \mid !p \\
\textit{Matches} & m & ::= & p \Rightarrow M \mid (m_1 \mid m_2) \mid \cdot
\end{array}
$$

An *extended case expression* has the form $\mathsf{case}\, M\, \mathsf{of}\, m$.

In the example of Booleans above, we gave a uniform definition for conditionals in terms of case. But can we define a function cond with arguments M, N_1 and N_2 which behaves like $\textbf{if}\, M\, \textbf{then}\, N_1\, \textbf{else}\, N_2$? The first difficulty is that the type of branches is generic. In order to avoid the complications of polymorphism, we uniformly define a whole family of functions cond_C types C. We go through some candidate types for cond_C and discuss why they may or may not be possible.

$\text{cond}_C : \mathbf{1} \oplus \mathbf{1} \multimap C \multimap C \multimap C$**.** This type means that both branches of the conditional (second and third argument) would be evaluated before being substituted in the definition of cond_C. Moreover, both must be used during the evaluation of the body, while intuitively only one branch should be used.

$\text{cond}_C : \mathbf{1} \oplus \mathbf{1} \multimap (!C) \multimap (!C) \multimap C$**.** This avoids evaluation of the branches, since they now can have the form $!N_1$ and $!N_2$, which are values. However, N_1 and N_2 can now no longer use linear variables.

$\text{cond}_C : \mathbf{1} \oplus \mathbf{1} \multimap C \to C \to C$**.** This is equivalent to the previous type and undesirable for the same reason.

$\text{cond}_C : \mathbf{1} \oplus \mathbf{1} \multimap (C \& C) \multimap C$**.** This type expresses that the second argument of type $C \& C$ is a pair $\langle N_1, N_2 \rangle$ such that exactly one component of this pair will be used. This expresses precisely the expected behavior and we define

$$\begin{array}{lcl}\mathrm{cond}_C & : & \mathbf{1} \oplus \mathbf{1} \multimap (C \& C) \multimap C \\ & = & \hat{\lambda} b{:}\mathbf{1} \oplus \mathbf{1}.\ \hat{\lambda} n{:}C \& C. \\ & & \quad \mathsf{case}\ b \\ & & \qquad \mathsf{of}\ \mathsf{inl} \star \Rightarrow \mathsf{fst}\, n \\ & & \qquad\ \mid \mathsf{inr} \star \Rightarrow \mathsf{snd}\, n \end{array}$$

which is linearly well-typed: b is used as the subject of the case and n is used in both branches of the case expression (which is additive).

As a first property of evaluation, we show that it is a strategy for β-reductions. That is, if $M \hookrightarrow v$ then M reduces to v in some number of β-reduction steps (possibly none), but not *vice versa*. For this we need a new judgment $M \longrightarrow_\beta^* M'$ is the congruent, reflexive, and transitive closure of the $M \longrightarrow_\beta M'$ relation. In other words, we extend β-reduction so it can be applied to an arbitrary subterm of M and then allow arbitrary sequences of reductions. The subject reduction property holds for this judgment as well.

Theorem 6.6 (Generalized Subject Reduction) *If* $\Gamma; \Delta \vdash M : A$ *and* $M \longrightarrow_\beta^* M'$ *then* $\Gamma; \Delta \vdash M' : A$.

Proof: See Exercise 6.8 □

Evaluation is related to β-reduction in that an expression reduces to its value.

Theorem 6.7 *If* $M \hookrightarrow v$ *then* $M \longrightarrow_\beta^* v$.

Proof: By induction on the structure of the derivation of $M \hookrightarrow v$. In each case we directly combine results obtained by appealing to the induction hypothesis using transitivity and congruence. □

The opposite is clearly false. For example,

$$\langle (\hat{\lambda} w{:}\mathbf{1}.\ w)\,\hat{}\,\star, \star\rangle \longrightarrow_\beta^* \langle \star, \star \rangle,$$

but

$$\langle (\hat{\lambda} w{:}\mathbf{1}.\ w)\,\hat{}\,\star, \star\rangle \hookrightarrow \langle (\hat{\lambda} w{:}\mathbf{1}.\ w)\,\hat{}\,\star, \star\rangle$$

and this is the only evaluation for the pair. However, if we limit the congruence rules to the components of $\otimes$, inl, inr, and all elimination constructs, the correspondence is exact (see Exercise 6.9). Type preservation is a simple consequence of the previous two theorems. See Exercise 6.10 for a direct proof.

Theorem 6.8 (Type Preservation) *If* $\cdot; \cdot \vdash M : A$ *and* $M \hookrightarrow v$ *then* $\cdot; \cdot \vdash v : A$.

Proof: By Theorem 6.7, $M \longrightarrow_\beta^* v$. Then the result follows by generalized subject reduction (Theorem 6.6). □

The final theorem of this section establishes the uniqueness of values.

Theorem 6.9 (Determinacy) *If $M \hookrightarrow v$ and $M \hookrightarrow v'$ then $v = v'$.*

Proof: By straightforward simultaneous induction on the structure of the two given derivations. For each for of M except case expressions there is exactly one inference rule which could be applied. For `case` we use the uniqueness of the value of the case subject to determine that the same rule must have been used in both derivations. □

We can also prove that evaluation of any closed, well-typed term M terminates in this fragment. We postpone the proof of this (Theorem 6.12) until we have seen further, more realistic, examples.

6.4 Recursive Types

The language so far lacks basic data types, such as natural numbers, integers, lists, trees, etc. Moreover, except for finitary ones such as booleans, they are not definable with the mechanism at our disposal so far. At this point we can follow two paths: one is to define each new data type in the same way we defined the logical connectives, that is, by introduction and elimination rules, carefully checking their local soundness and completeness. The other is to enrich the language with a general mechanism for defining such new types. Again, this can be done in different ways, using either *inductive types* which allow us to maintain a clean connection between propositions and types, or *recursive types* which are more general, but break the correspondence to logic. Since we are mostly interested in programming here, we chose the latter path.

Recall that we defined the booleans as $\mathbf{1} \oplus \mathbf{1}$. It is easy to show by the definition of values, that there are exactly two values of this type, to which we can arbitrarily assign true and false. A finite type with n values can be defined as the disjoint sum of n observable singleton types, $\mathbf{1} \oplus \cdots \oplus \mathbf{1}$. The natural numbers would be $\mathbf{1} \oplus \mathbf{1} \oplus \cdots$, except that this type is infinite. We can express it finitely as a recursive type $\mu\alpha.\ \mathbf{1} \oplus \alpha$. Intuitively, the meaning of this type should be invariant under unrolling of the recursion. That is,

$$\begin{array}{rcl}
\mathrm{nat} & = & \mu\alpha.\ \mathbf{1} \oplus \alpha \\
 & \cong & [(\mu\alpha.\ \mathbf{1} \oplus \alpha)/\alpha]\mathbf{1} \oplus \alpha \\
 & = & \mathbf{1} \oplus \mu\alpha.\ \mathbf{1} \oplus \alpha \\
 & = & \mathbf{1} \oplus \mathrm{nat}
\end{array}$$

which is the expected recursive definition for the type of natural numbers.

In functional languages such as ML or Haskell, recursive type definitions are not directly available, but the results of elaborating syntactically more pleaseant definitions. In addition, recursive type definitions are *generative*, that is, they generate new constructors and types every time they are invoked. This is of great practical value, but the underlying type theory can be seen as simple recursive types combined with a mechanism for generativity. Here, we will only treat the issue of recursive types.

Even though recursive types do not admit a logical interpretation as propositions, we can still define a term calculus using introduction and elimination rules, including local reduction and expansions. In order maintain the property that a term has a unique type, we annotate the introduction constant fold with the recursive type itself.

$$\frac{\Gamma; \Delta \vdash M : [\mu\alpha.\ A/\alpha]A}{\Gamma; \Delta \vdash \mathrm{fold}^{\mu\alpha.\ A} M : \mu\alpha.\ A}\,\mu\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash M : \mu\alpha.\ A}{\Gamma; \Delta \vdash \mathrm{unfold}\, M : [\mu\alpha.\ A/\alpha]A}\,\mu\mathrm{E}$$

The local reduction and expansions, expressed on the terms.

$$\begin{array}{rcl} \mathrm{unfold}\,\mathrm{fold}^{\mu\alpha.\ A}\, M & \longrightarrow_\beta & M \\ M : \mu\alpha.\ A & \longrightarrow_\eta & \mathrm{fold}^{\mu\alpha.\ A}\,(\mathrm{unfold}\, M) \end{array}$$

It is easy to see that uniqess of types and subject reduction remain valid properties (see Exercise 6.11). There are also formulation of recursive types where the term M in the premiss and conclusion is the same, that is, there are no explicit constructor and destructors for recursive types. This leads to more concise programs, but significantly more complicated type-checking (see Exercise 6.12).

We would like recursive types to represent data types. Therefore the values of recursive type must be of the form $\mathrm{fold}^{\mu\alpha.\ A}\, v$ for values v—otherwise data values would not be observable.

$$\frac{M\ \mathit{Value}}{\mathrm{fold}^{\mu\alpha.\ A}\, M\ \mathit{Value}}\,\mu\mathrm{val}$$

$$\frac{M \hookrightarrow v}{\mathrm{fold}^{\mu\alpha.\ A}\, M \hookrightarrow \mathrm{fold}^{\mu\alpha.\ A}\, v}\,\mu\mathrm{Iv} \qquad \frac{M \hookrightarrow \mathrm{fold}^{\mu\alpha.\ A}\, v}{\mathrm{unfold}\, M \hookrightarrow v}\,\mu\mathrm{Ev}$$

In order to write interesting programs simply, it is useful to have a general recursion operator $\mathbf{fix}\, u{:}A.\ M$ at the level of terms. It is not associated with an type constructor and simply unrolls its definition once when executed. In the typing rule we have to be careful: since the number on unrollings generally unpredictable, no linear variables are permitted to occur free in the body of a recursive definition. Moreover, the recursive function itself may be called arbitrarily many times—one of the characteristics of recursion. Therefore its uses are unrestricted.

$$\frac{(\Gamma, u{:}A); \cdot \vdash M : A}{\Gamma; \cdot \vdash \mathbf{fix}\, u{:}A.\ M : A}\,\mathbf{fix}$$

The operator does not introduce any new values, and one new evaluation rules which unrolls the recursion.

$$\frac{[\mathbf{fix}\, u{:}A.\ M/u]M \hookrightarrow v}{\mathbf{fix}\, u{:}A.\ M \hookrightarrow v}\,\mathbf{fix}\mathrm{v}$$

In order to guarantee subject reduction, the type of whole expression, the body M of the fixpoint expression, and the bound variable u must all have the same type A. This is enforced in the typing rules.

We now consider a few examples of recursive types and some example programs.

Natural Numbers.

$$
\begin{array}{rcl}
\mathrm{nat} & = & \mu\alpha.\, \mathbf{1} \oplus \alpha \\
\mathrm{zero} & : & \mathrm{nat} \\
 & = & \mathrm{fold}^{\mathrm{nat}}\, (\mathsf{inl}^{\mathrm{nat}}\, \star) \\
\mathrm{succ} & : & \mathrm{nat} \multimap \mathrm{nat} \\
 & = & \hat{\lambda} x{:}\mathrm{nat}.\ \mathrm{fold}^{\mathrm{nat}}\, (\mathsf{inr}^{\mathbf{1}}\, x)
\end{array}
$$

With this definition, the addition function for natural numbers is linear in both argument.

$$
\begin{array}{rcl}
\mathrm{plus} & : & \mathrm{nat} \multimap \mathrm{nat} \multimap \mathrm{nat} \\
 & = & \mathbf{fix}\, p{:}\mathrm{nat} \multimap \mathrm{nat} \multimap \mathrm{nat}. \\
 & & \quad \hat{\lambda} x{:}\mathrm{nat}.\ \hat{\lambda} y{:}\mathrm{nat}.\ \mathsf{case}\ \mathrm{unfold}\, x \\
 & & \qquad\qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow y \\
 & & \qquad\qquad\qquad\qquad\ \mid \mathsf{inr}\, x' \Rightarrow \mathrm{succ}\,\hat{}\,(p\,\hat{}\,x'\,\hat{}\,y)
\end{array}
$$

It is easy to ascertain that this definition is well-typed: x occurs as the case subject, y in both branches, and x' in the recursive call to p. On the other hand, the natural definition for multiplication is *not* linear, since the second argument is used twice in one branch of the case statement and not at all in the other.

$$
\begin{array}{rcl}
\mathrm{mult} & : & \mathrm{nat} \multimap \mathrm{nat} \rightarrow \mathrm{nat} \\
 & = & \mathbf{fix}\, m{:}\mathrm{nat} \multimap \mathrm{nat} \rightarrow \mathrm{nat} \\
 & & \quad \hat{\lambda} x{:}\mathrm{nat}.\ \lambda y{:}\mathrm{nat}.\ \mathsf{case}\ \mathrm{unfold}\, x \\
 & & \qquad\qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow \mathrm{zero} \\
 & & \qquad\qquad\qquad\qquad\ \mid \mathsf{inr}\, x' \Rightarrow \mathrm{plus}\,\hat{}\,(m\,\hat{}\,x'\, y)\,\hat{}\,y
\end{array}
$$

Interestingly, there is also a linear definition of mult (see Exercise 6.13), but its operational behavior is quite different. This is because we can explicitly copy and delete natural numbers, and even make them available in an unrestricted way.

$$
\begin{array}{rcl}
\text{copy} & : & \text{nat} \multimap \text{nat} \otimes \text{nat} \\
& = & \textbf{fix}\, c{:}\text{nat} \multimap \text{nat} \otimes \text{nat} \\
& & \quad \hat{\lambda} x{:}\text{nat}.\ \mathsf{case}\ \text{unfold}\, x \\
& & \qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow \text{zero} \otimes \text{zero} \\
& & \qquad\qquad\qquad \mid \mathsf{inr}\, x' \Rightarrow \mathsf{let}\, x_1' \otimes x_2' = c \,\hat{}\, x'\, \mathsf{in}\, (\text{succ} \,\hat{}\, x_1') \otimes (\text{succ} \,\hat{}\, x_2') \\
\text{delete} & : & \text{nat} \multimap \mathbf{1} \\
& = & \textbf{fix}\, d{:}\text{nat} \multimap \mathbf{1} \\
& & \quad \hat{\lambda} x{:}\text{nat}.\ \mathsf{case}\ \text{unfold}\, x \\
& & \qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow \mathbf{1} \\
& & \qquad\qquad\qquad \mid \mathsf{inr}\, x' \Rightarrow \mathsf{let}\, \star = d \,\hat{}\, x'\, \mathsf{in}\, \mathbf{1} \\
\text{promote} & : & \text{nat} \multimap\, !\text{nat} \\
& = & \textbf{fix}\, p{:}\text{nat} \multimap\, !\text{nat} \\
& & \quad \hat{\lambda} x{:}\text{nat}.\ \mathsf{case}\ \text{unfold}\, x \\
& & \qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow\, !\text{zero} \\
& & \qquad\qquad\qquad \mid \mathsf{inr}\, x' \Rightarrow \mathsf{let}\, !u' = p \,\hat{}\, x'\, \mathsf{in}\, !(\text{succ}\, u')
\end{array}
$$

Lazy Natural Numbers. Lazy natural numbers are a simple example of *lazy data types* which contain unevaluated expressions. Lazy data types are useful in applications with potentially infinite data such as streams. We encode such lazy data types by using the $!A$ type constructor.

$$
\begin{array}{rcl}
\text{lnat} & = & \mu\alpha.\, !(\mathbf{1} \oplus \alpha) \\
\text{lzero} & : & \text{lnat} \\
& = & \text{fold}^{\text{lnat}}\, !\, (\mathsf{inl}^{\text{lnat}}\, \star) \\
\text{lsucc} & : & \text{lnat} \to \text{lnat} \\
& = & \lambda u{:}\text{lnat}.\ \text{fold}^{\text{lnat}}\, !\, (\mathsf{inr}^{\mathbf{1}}\, u)
\end{array}
$$

There is also a linear version of successor of type, $\text{lnat} \multimap \text{lnat}$, but it is not as natural since it evaluates its argument just to build another lazy natural number.

$$
\begin{array}{rcl}
\text{lsucc}' & : & \text{lnat} \multimap \text{lnat} \\
& = & \hat{\lambda} x{:}\text{lnat}.\ \mathsf{let}\, !u = \text{unfold}\, x\, \mathsf{in}\, \text{fold}^{\text{lnat}}\, (!\, \mathsf{inr}^{\mathbf{1}}\, (\text{fold}^{\text{lnat}}\, (!u)))
\end{array}
$$

The "infinite" number number ω can be defined by using the fixpoint operator. We can either use lsucc as defined above, or define it directly.

$$
\begin{array}{rcl}
\omega & : & \text{lnat} \\
& = & \textbf{fix}\, u{:}\text{lnat}.\ \text{lsucc}\, u \\
& \cong & \textbf{fix}\, u{:}\text{lnat}.\ \text{fold}^{\text{lnat}}\, !\, (\mathsf{inr}^{\mathbf{1}}\, u)
\end{array}
$$

Note that lazy natural numbers are not directly observable (except for the $\text{fold}^{\text{lnat}}$), so we have to decompose and examine the structure of a lazy natural number successor by successor, or we can convert it to an observable natural number (which might not terminate).

$$\begin{array}{lcl}
\mathrm{toNat} & : & \mathrm{lnat} \multimap \mathrm{nat} \\
& = & \mathbf{fix}\, t{:}\mathrm{lnat} \multimap \mathrm{nat} \\
& & \quad \hat{\lambda} x{:}\mathrm{lnat}.\ \mathsf{case}\ \mathrm{unfold}\, x \\
& & \qquad\qquad \mathsf{of}\ !\mathsf{inl}^{\mathrm{lnat}} \star \Rightarrow \mathrm{zero} \\
& & \qquad\qquad \mid\ !\mathsf{inr}^{\mathbf{1}}\, x' \Rightarrow \mathrm{succ}\,\hat{}\,(t\,\hat{}\,x')
\end{array}$$

Lists. To avoid issues of polymorphism, we define a family of data types list_A for an arbitrary type A.

$$\begin{array}{rcl}
\mathrm{list}_A & = & \mu\alpha.\ \mathbf{1} \oplus (A \otimes \alpha) \\
\mathrm{nil}_A & : & \mathrm{list}_A \\
& = & \mathrm{fold}^{\mathrm{list}_A} (\mathsf{inl}^{\mathrm{list}_A} \star) \\
\mathrm{cons}_A & : & A \otimes \mathrm{list}_A \multimap \mathrm{list}_A \\
& = & \hat{\lambda} p{:}A \otimes \mathrm{list}_A.\ \mathrm{fold}^{\mathrm{list}_A} (\mathsf{inr}^{\mathbf{1}}\, p)
\end{array}$$

We can easily program simple functions such as append and reverse which are linear in their arguments. We show here reverse; for other examples see Exercise 6.14. we define an auxiliary tail-recursive function rev which moves element from it first argument to its second.

$$\begin{array}{rcl}
\mathrm{rev}_A & : & \mathrm{list}_A \multimap \mathrm{list}_A \multimap \mathrm{list}_A \\
& = & \mathbf{fix}\, r{:}\mathrm{list}_A \multimap \mathrm{list}_A \multimap \mathrm{list}_A \\
& & \quad \hat{\lambda} l{:}\mathrm{list}_A.\ \hat{\lambda} k{:}\mathrm{list}_A. \\
& & \qquad \mathsf{case}\ \mathrm{unfold}\, l \\
& & \qquad\quad \mathsf{of}\ \mathsf{inl}^{A \otimes \mathrm{list}_A} \star \Rightarrow k \\
& & \qquad\quad \mid \mathsf{inr}^{\mathbf{1}}\, (x \otimes l') \Rightarrow r\,\hat{}\,l'\,\hat{}\,(\mathrm{cons}_A\, (x \otimes k)) \\
\mathrm{reverse}_A & : & \mathrm{list}_A \multimap \mathrm{list}_A \\
& = & \hat{\lambda} l{:}\mathrm{list}_A.\ \mathrm{rev}\,\hat{}\,l\,\hat{}\,\mathrm{nil}_A
\end{array}$$

To make definitions like this a bit easier, we can also define a case for lists, in analogy with the conditional for booleans. It is a family indexed by the type of list elements A and the type of the result of the conditional C.

$$\begin{array}{rcl}
\mathrm{listCase}_{A,C} & : & \mathrm{list}_A \multimap (C \,\&\, (A \otimes \mathrm{list}_A \multimap C)) \multimap C \\
& = & \hat{\lambda} l{:}\mathrm{list_A}.\ \hat{\lambda} n{:}C \,\&\, (A \otimes \mathrm{list}_A \multimap C). \\
& & \qquad \mathsf{case}\ \mathrm{unfold} l \\
& & \qquad\quad \mathsf{of}\ \mathsf{inl}^{A \otimes \mathrm{list}_A} \star \Rightarrow \mathsf{fst}\, n \\
& & \qquad\quad \mid \mathsf{inr}^{\mathbf{1}}\, p \Rightarrow (\mathsf{snd}\, n)\,\hat{}\,p
\end{array}$$

Lazy Lists. There are various forms of lazy lists, depending of which evaluation is postponed.

$\mathrm{llist}^1_A = \mu\alpha.\ !(\mathbf{1} \oplus (A \otimes \alpha))$. This is perhaps the canonical lazy lists, in which we can observe neither head nor tail.

$\text{llist}^2_A = \mu\alpha.\, \mathbf{1} \oplus {!}(A \otimes \alpha)$. Here we can observe directly if the list is empty or not, but not the head or tail which remains unevaluated.

$\text{llist}^3_A = \mu\alpha.\, \mathbf{1} \oplus (A \otimes {!}\alpha)$. Here we can observe directly if the list is empty or not, and the head of the list is non-empty. However, we cannot see the tail.

$\text{llist}^4_A = \mu\alpha.\, \mathbf{1} \oplus ({!}A \otimes \alpha)$. Here the list is always eager, but the elements are lazy. This is the same as $\text{list}_{!A}$.

$\text{llist}^5_A = \mu\alpha.\, \mathbf{1} \oplus (A \& \alpha)$. Here we can see if the list is empty or not, but we can access only either the head or tail of list, but not both.

$\text{infStream}_A = \mu\alpha.\, {!}(A \otimes \alpha)$. This is the type of infinite streams, that is, lazy lists with no nil constructor.

Functions such as append, map, etc. can also be written for lazy lists (see Exercise 6.15).

Other types, such as trees of various kinds, are also easily represented using similar ideas. However, the recursive types (even without the presence of the fixpoint operator on terms) introduce terms which have no normal form. In the pure, untyped λ-calculus, the classical examples of a term with no normal form is $(\lambda x.\ x\, x)\, (\lambda x.\ x\, x)$ which β-reduces to itself in one step. In the our typed λ-calculus (linear or intuitionistic) this cannot be assigned a type, because x is used as an argument to itself. However, with recursive types (and the fold and unfold constructors) we can give a type to a version of this term which β-reduces to itself in two steps.

$$
\begin{array}{rcl}
\Omega & = & \mu\alpha.\, \alpha \to \alpha \\
\omega & : & \Omega \to \Omega \\
 & = & \lambda x{:}\Omega.\ (\text{unfold}\, x)\, x
\end{array}
$$

Then

$$
\begin{array}{l}
\omega\, (\text{fold}^{\Omega}\, \omega) \\
\quad \longrightarrow_\beta (\text{unfold}\, (\text{fold}^{\Omega}\, \omega))\, (\text{fold}^{\Omega}\, \omega) \\
\quad \longrightarrow_\beta \omega\, (\text{fold}^{\Omega}\, \omega).
\end{array}
$$

At teach step we applied the only possible β-reduction and therefore the term can have no normal form. An attempt to evaluate this term will also fail, resulting in an infinite regression (see Exercise 6.16).

6.5 Termination

As the example at the end of the previous section shows, unrestricted recursive types destroy the normalization property. This also means it is impossible to give all recursive types a logical interpretation. When we examine the inference rules we notice that recursive types are *impredicative*: the binder $\mu\alpha$ in $\mu\alpha.\ A$

ranges over the whole type. This means in the introduction rule, the type in the premiss $[\mu\alpha.\ A/\alpha]A$ generally will be larger than the type $\mu\alpha.\ A$ in the conclusion. That alone is not responsible for non-termination: there are other type disciplines such as the polymorphic λ-calculus which retain a logical interpretation and termination, yet are impredicative.

In this section we focus on the property that all well-typed terms in the linear λ-calculus *without* recursive types and fixpoint operators evaluate to a value. This is related to the normalization theorem for natural deductions (Theorem 3.10): if $\Gamma; \Delta \vdash A$ then $\Gamma; \Delta \vdash A \uparrow$. We proved this by a rather circuitous route: unrestricted natural deductions can be translated to sequent derivations with cut from which we can eliminate cut and translate the result cut-free derivation back to a noraml natural deduction.

Here, we prove directly that every term evaluates using the proof technique of *logical relations* [Sta85] also called *Tait's method* [Tai67]. Because of the importance of this technique, we spend some time motivating its form. Our ultimate goal is to prove:

If $\cdot;\cdot \vdash M : A$ *then* $M \hookrightarrow V$ *for some value* V.

The first natural attempt would be to prove this by induction on the typing derivation. Surprisingly, case for $\multimap$I works, even though we cannot apply the inductive hypothesis, since every linear λ-abstraction immediately evaluates to itself.

In the case for $\multimap$E, however, we find that we cannot complete the proof. Let us examine why.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \cdot;\cdot \vdash M_1 : A_2 \multimap A_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \cdot;\cdot \vdash M_2 : A_2\end{array}}{\cdot;\cdot \vdash M_1 \hat{\ } M_2 : A_1} \multimap\mathrm{E}.$

We can make the following inferences.

$M_1 \hookrightarrow V_1$ for some V_1	By ind. hyp. on $\mathcal{D}_1$
$V_1 = \hat{\lambda} x{:}A_2.\ M_1'$	By type preservation and inversion
$M_2 \hookrightarrow V_2$ for some V_2	By ind. hyp. on $\mathcal{D}_2$

At this point we cannot proceed: we need a derivation of

$[V_2/x]M_1' \hookrightarrow V$ *for some* V

to complete the derivation of $M_1 \hat{\ } M_2 \hookrightarrow V$. Unfortunately, the induction hypothesis does not tell us anything about $[V_2/x]M_1'$. Basically, we need to extend it so it makes a statement about the *result* of evaluation ($\hat{\lambda} x{:}A_2.\ M_1'$, in this case).

Sticking to the case of linear application for the moment, we call a term M "good" if it evaluates to a "good" value V. A value V is "good" if it is a function $\hat{\lambda} x{:}A_2.\ M_1'$ and if substituting a "good" value V_2 for x in M_1' results in a "good" term. Note that this is not a proper definition, since to see if V is "good" we may need to substitute any "good" value V_2 into it, possibly including V itself. We can make this definition inductive if we observe that the value V_2 will be of type A_2, while the value V we are testing has type $A_2 \multimap A_1$, and that the resulting term as type A_1. That is, we can fashion a definition which is inductive on the structure of the *type*. Instead of saying "good" we say $M \in \|A\|$ and $v \in |A|$. Still restricting ourselves to linear implication only, we define:

$$\begin{array}{rcl} M \in \|A\| & \text{iff} & M \hookrightarrow V \text{ and } V \in |A| \\ M \in |A_2 \multimap A_1| & \text{iff} & M = \hat{\lambda}x{:}A_2.\ M_1 \text{ and } [V_2/x]M_1 \in \|A_1\| \text{ for any } V_2 \in |A_2| \end{array}$$

From $M \in \|A\|$ we can immediately infer $M \hookrightarrow V$ for some V, so when proving that $\cdot;\cdot \vdash M : A$ implies $M \in \|A\|$ we do indeed have a much stronger induction hypothesis.

While the case for application now goes through, the case for linear λ-abstraction fails, since we cannot prove the stronger property for the value.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \cdot; x{:}A_2 \vdash M_1 : A_1\end{array}}{\cdot;\cdot \vdash \hat{\lambda}x{:}A_2.\ M_1 : A_2 \multimap A_1} \multimap\text{I}.$

Then $\hat{\lambda}x{:}A_2.\ M_1 \hookrightarrow \hat{\lambda}x{:}A_2.\ M_1$ and it remains to show that for every $V_2 \in |A_2|$, $[V_2/x]M_2 \in \|A_1\|$.

This last statement should follow from the induction hypothesis, but presently it is too weak since it only allows for closed terms. The generalization which suggests itself from this case (ignoring the unrestricted context for now) is:

> If $\Delta \vdash M : A$, then for any substitution θ which maps the linear variables $x{:}A$ in Δ to values $V \in |A|$, $[\theta]M \in \|A\|$.

This generalization indeed works after we also account for the unrestricted context. During evaluation we substitute values for linear variables and expressions for unrestricted variables. Therefore, the substitutions we must consider for the induction hypothesis have to behave accordingly.

$$\begin{array}{rrcl} \textit{Unrestricted Substitution} & \eta & ::= & \cdot \mid \eta, M/u \\ \textit{Linear Substitution} & \theta & ::= & \cdot \mid \theta, V/x \end{array}$$

We write $[\eta;\theta]M$ for the simultaneous application of the substitutions η and θ to M. For our purposes here, the values and terms in the substitutions are always closed, but we do not need to postulate this explicitly. Instead, we only deal with substitution satisfying the property necessary for the generalization of the induction hypothesis.

$$\begin{array}{rcl} \theta \in |\Delta| & \text{iff} & [\theta]x \in |A| \text{ for every } x{:}A \text{ in } \Delta \\ \eta \in \|\Gamma\| & \text{iff} & [\eta]u \in \|A\| \text{ for every } u{:}A \text{ in } \Gamma \end{array}$$

We need just one more lemma, namely that values evaluate to themselves.

Lemma 6.10 (Value Evaluation) *For any value v, $v \hookrightarrow v$*

Proof: See Exercise 6.18. $\square$

Now we have all ingredients to state the main lemma in the proof of termination, the so called *logical relations lemma* [Sta85]. The "logical relations" are $\|A\|$ and $|A|$, seen as unary relations, that is, predicates, on terms and values, respectively. They are "logical" since they are defined by induction on the structure of the type A, which corresponds to a proposition under the Curry-Howard isomorphism.

Lemma 6.11 (Logical Relations) *If $\Gamma; \Delta \vdash M : A$, $\eta \in \|\Gamma\|$ and $\theta \in |\Delta|$ then $[\eta;\theta]M \in \|A\|$.*

Before showing the proof, we extend the definition of the logical relations to all the types we have been considering.

$$
\begin{array}{rcl}
M \in \|A\| & \text{iff} & M \hookrightarrow V \text{ and } V \in |A| \\
V \in |A_2 \multimap A_1| & \text{iff} & V = \hat{\lambda} x{:}A_2.\ M_1 \text{ and } [V_2/x]M_1 \in \|A_1\| \text{ for any } V_2 \in |A_2| \\
V \in |A_1 \otimes A_2| & \text{iff} & V = V_1 \otimes V_2 \text{ where } V_1 \in |A_1| \text{ and } V_2 \in |A_2| \\
V \in |\mathbf{1}| & \text{iff} & V = \star \\
V \in |A_1 \& A_2| & \text{iff} & V = \langle M_1, M_2 \rangle \text{ where } M_1 \in \|A_1\| \text{ and } M_2 \in \|A_2\| \\
V \in |\top| & \text{iff} & V = \langle\,\rangle \\
V \in |A_1 \& A_2| & \text{iff} & \text{either } V = \mathsf{inl}^{A_2}\, V_1 \text{ and } V_1 \in |A_1|, \\
 & & \text{or } V = \mathsf{inr}^{A_1}\, V_2 \text{ and } V_2 \in |A_2| \\
V \in |\mathbf{0}| & & \text{never} \\
V \in |!A| & \text{iff} & V = !M \text{ and } M \in \|A\| \\
V \in |A_2 \to A_1| & \text{iff} & V = \lambda u{:}A_2.\ M_1 \text{ and } [M_2/u]M_1 \in \|A_1\| \\
 & & \text{for any } M_2 \in \|A_2\|
\end{array}
$$

These definitions are motivated directly from the form of values in the language. One can easily see that it is indeed inductive on the structure of the type. If we tried to add recursive types in a similar way, the proof below would still go through, except that the definition of the logical relation would no longer be well-founded.

Proof: (of the logical relations lemma 6.11). The proof proceeds by induction on the structure of the typing derivation $\mathcal{D} :: (\Gamma; \Delta \vdash M : A)$. We show three cases—all others are similar.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma; \Delta \vdash M_1 : A_2 \multimap A_1 & \Gamma; \Delta \vdash M_2 : A_2 \end{array}}{\Gamma; \Delta \vdash M_1 \,\hat{}\, M_2 : A_1} \multimap E.$

$\eta \in \|\Gamma\|$	Assumption
$\theta \in \|\Delta\|$	Assumption
$[\eta;\theta]M_1 \in \|A_2 \multimap A_1\|$	By ind. hyp. on $\mathcal{D}_1$

$\mathcal{E}_1 :: ([\eta;\theta]M_1 \hookrightarrow V_1)$ and $V_1 \in \lvert A_2 \multimap A_1\rvert$	By definition of $\lVert A_2 \multimap A_1\rVert$
$V_1 = \hat{\lambda} x{:}A_1.\, M_1'$ and	
$[V_2/x]M_1' \in \lVert A_1\rVert$ for any $V_2 \in \lvert A_2\rvert$	By definition of $\lvert A_2 \multimap A_1\rvert$
$[\eta;\theta]M_2 \in \lVert A_2\rVert$	By ind. hyp. on $\mathcal{D}_2$
$\mathcal{E}_2 :: ([\eta;\theta]M_2 \hookrightarrow V_2)$ and $V_2 \in \lvert A_2\rvert$	By definition of $\lVert A_2\rVert$
$[V_2/x]M_1' \in \lVert A_1\rVert$	Since $V_2 \in \lvert A_2\rvert$
$\mathcal{E}_3 :: ([V_2/x]M_1' \hookrightarrow V)$ and $V \in \lvert A_1\rvert$	by definition of $\lVert A_1\rVert$
$\mathcal{E} :: ([\eta;\theta](M_1 \hat{\ } M_2) \hookrightarrow V)$	by $\multimap$Ev from $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$
$[\eta;\theta](M_1 \hat{\ } M_2) \in \lVert A_1\rVert$	by definition of $\lVert A_1\rVert$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma;(\Delta, x{:}A_2) \vdash M_1 : A_1\end{array}}{\Gamma;\Delta \vdash \hat{\lambda} x{:}A_2.\, M_1 : A_2 \multimap A_1}\multimap\mathrm{I}.$

$\eta \in \lVert\Gamma\rVert$	by assumption
$\theta \in \lvert\Delta\rvert$	by assumption
$\mathcal{E} :: ([\eta;\theta](\hat{\lambda} x{:}A_2.\, M_1) \hookrightarrow [\eta;\theta](\hat{\lambda} x{:}A_2.\, M_1))$	by $\multimap$Iv
$V_2 \in \lvert A_2\rvert$	assumption
$(\theta, V_2/x) \in \lvert\Delta, x{:}A_2\rvert$	by definition of $\lvert\Delta\rvert$
$[\eta;(\theta, V_2/x)]M_1 \in \lVert A_1\rVert$	by ind. hyp. on $\mathcal{D}_1$
$[V_2/x]([\eta;(\theta, x/x)]M_1) \in \lVert A_1\rVert$	by properties of substitution
$(\hat{\lambda} x{:}A_2.\, [\eta;(\theta, x/x)]M_1) \in \lvert A_2 \multimap A_1\rvert$	by definition of $\lvert A_2 \multimap A_1\rvert$
$[\eta;\theta](\hat{\lambda} x{:}A_2.\, M_1) \in \lvert A_2 \multimap A_1\rvert$	by properties of substitution
$[\eta;\theta](\hat{\lambda} x{:}A_2.\, M_1) \in \lVert A_2 \multimap A_1\rVert$	by definition of $\lVert A_2 \multimap A_1\rVert$

Case: $\mathcal{D} = \dfrac{}{\Gamma; x{:}A \vdash x : A}\mathsf{hyp}$

$\theta \in \lvert\cdot, x{:}A\rvert$	by assumption
$[\theta]x \in \lvert A\rvert$	by definition of $\lvert\cdot, x{:}A\rvert$
$\mathcal{E} :: ([\eta;\theta]x \hookrightarrow [\eta;\theta]x)$	by Lemma 6.10
$[\eta;\theta]x \in \lVert A\rVert$	by definition of $\lVert A\rVert$

$\square$

The termination theorem follows directly from the logical relations lemma. Note that the theorem excludes recursive types and the fixpoint operator by a general assumption for this section.

Theorem 6.12 (Termination) *If* $\cdot;\cdot \vdash M : A$ *then* $M \hookrightarrow V$ *for some* V.

Proof: We have $\cdot \in \lVert\cdot\rVert$ and $\cdot \in \lvert\cdot\rvert$ since the conditions are vacuously satisfied. Therefore, by the logical relations lemma 6.11, $[\cdot;\cdot]M \in \lVert A\rVert$. By the definition of $\lVert A\rVert$ and the observation that $[\cdot;\cdot]M = M$, we conclude that $M \hookrightarrow V$ for some V. $\square$

6.6 Exercises

Exercise 6.1 Prove that if $\Gamma; \Delta \vdash M : A$ and $\Gamma; \Delta \vdash M : A'$ then $A = A'$.

Exercise 6.2 A function in a functional programming language is called *strict* if it is guaranteed to use its argument. Strictness is an important concept in the implementation of lazy functional languages, since a strict function can evaluate its argument eagerly, avoiding the overhead of postponing its evaluation and later memoizing its result.

In this exercise we design a λ-calculus suitable as the core of a functional language which makes strictness explicit at the level of types. Your calculus should contain an unrestricted function type $A \rightarrow B$, a strict function type $A \twoheadrightarrow B$, a vacuous function type $A \dashrightarrow B$, a full complement of operators refining product and disjoint sum types as for the linear λ-calculus, and a modal operator to internalize the notion of closed term as in the linear λ-calculus. Your calculus should not contain quantifiers.

1. Show the introduction and elimination rules for all types, including their proof terms.

2. Given the reduction and expansions on the proof terms.

3. State (without proof) the valid substitution principles.

4. If possible, give a translation from types and terms in the strict λ-calculus to types and terms in the linear λ-calculus such that a strict term is well-typed if and only if its linear translation is well-typed (in an appropriately translated context).

5. Either sketch the correctness proof for your translation in each direction by giving the generalization (if necessary) and a few representative cases, or give an informal argument why such a translation is not possible.

Exercise 6.3 Give an example which shows that the substitution $[M/w]N$ must be capture-avoiding in order to be meaningful. *Variable capture* is a situation where a bound variable w' in N occurs free in M, and w occurs in the scope of w'. A similar definition applies to unrestricted variables.

Exercise 6.4 Give a counterexample to the conjecture that if $M \longrightarrow_\beta M'$ and $\Gamma; \Delta \vdash M' : A$ then $\Gamma; \Delta \vdash M : A$. Also, either prove or find a counterexample to the claim that if $M \longrightarrow_\eta M'$ and $\Gamma; \Delta \vdash M' : A$ then $\Gamma; \Delta \vdash M : A$.

Exercise 6.5 The proof term assignment for sequent calculus identifies many distinct derivations, mapping them to the same natural deduction proof terms. Design an alternative system of proof terms from which the sequent derivation can be reconstructed uniquely (up to weakening of unrestricted hypotheses and absorption of linear hypotheses in the $\top$R rule).

1. Write out the term assignment rules for all propositional connectives.

2. Give a calculus of reductions which corresponds to the initial and principal reductions in the proof of admissibility of cut.

3. Show the reduction rule for the dereliction cut.

4. Show the reduction rules for the left and right commutative cuts.

5. Sketch the proof of the subject reduction properties for your reduction rules, giving a few critical cases.

6. Write a translation judgment $S \Longrightarrow M$ from faithful sequent calculus terms to natural deduction terms.

7. Sketch the proof of type preservation for your translation, showing a few critical cases.

Exercise 6.6 Supply the missing rules for $\oplus$E in the definition of the judgment $\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A$ and show the corresponding cases in the proof of Lemma 6.4.

Exercise 6.7 In this exercise we explore the syntactic expansion of *extended case expressions* of the form $\mathsf{case}\, M \,\mathsf{of}\, m$.

1. Define a judgment which checks if an extended case expression is valid. This is likely to require some auxiliary judgments. You must verify that the cases are exhaustive, circumscribe the legal patterns, and check that the overall expression is linearly well-typed.

2. Define a judgment which relates an extended case expression to its expansion in terms of the primitive let, case, and abort constructs in the linear λ-calculus.

3. Prove that an extended case expression which is valid according to your criteria can be expanded to a well-typed linear λ-term.

4. Define an operational semantics directly on extended case expressions.

5. Prove that your direct operational semantics is correct on valid patterns with respect to the translational semantics from questions 2.

Exercise 6.8 Define the judgment $M \longrightarrow_\beta^* M'$ via inference rules. The rules should directly express that it is the congruent, reflexive and transitive closure of the β-reduction judgment $M \longrightarrow_\beta M'$. Then prove the generalized subject reduction theorem 6.6 for your judgment. You do not need to show all cases, but you should carefully state your induction hypothesis in sufficient generality and give a few critical parts of the proof.

Exercise 6.9 Define *weak β-reduction* as allows simple β-reduction under $\otimes$, inl, and inr constructs and in all components of the elimination form. Show that if M weakly reduces to a value v then $M \hookrightarrow v$.

Exercise 6.10 Prove type preservation (Theorem 6.8) directly by induction on the structure of the evaluation derivation, using the substitution lemma 6.2 as necessary, but without appeal to subject reduction.

Exercise 6.11 Prove the subject reduction and expansion properties for recursive type computation rules.

Exercise 6.12 [*An exercise exploring the use of type conversion rules without explicit term constructors.*]

Exercise 6.13 Define a linear multiplication function mult : $\mathrm{nat} \multimap \mathrm{nat} \multimap \mathrm{nat}$ using the functions copy and delete.

Exercise 6.14 Defined the following functions on lists. Always explicitly state the type, which should be the most natural type of the function.

1. append to append two lists.
2. concat to append all the lists in a list of lists.
3. map to map a function f over the elements of a list. The result of mapping f over the list $x_1, x_2, \ldots, x_n$ should be the list $f(x_1), f(x_2), \ldots f(x_n)$, where you should decide if the application of f to its argument should be linear or not.
4. foldr to reduce a list by a function f. The result of folding f over a list $x_1, x_2, \ldots x_n$ should be the list $f(x_1, f(x_2, \ldots, f(x_n, \mathit{init})))$, where *init* is an initial value given as argument to foldr. You should decide if the application of f to its argument should be linear or not.
5. copy, delete, and promote.

Exercise 6.15 For one of the form of lazy lists on Page 130, define the functions from Exercise 6.14 plus a function toList which converts the lazy to an eager list (and may therefore not terminate if the given lazy lists is infinite). Make sure that your functions exhibit the correct amount of laziness. For example, a map function applied to a lazy list should not carry out any non-trivial computation until the result is examined.

Further for your choice of lazy list, define the infinite lazy list of eager natural numbers $0, 1, 2, \ldots$.

Exercise 6.16 Prove that there is no term v such that $\omega\,(\mathrm{fold}^{\Omega}\,\omega) \hookrightarrow v$.

Exercise 6.17 [*An exercise about the definability of fixpoint operators at various type.*]

Exercise 6.18 Prove Lemma 6.10 which states that all values evaluate to themselves.

Exercise 6.19 In this exercise we explore strictness as a derived, rather than a primitive concept. Recall that a function is *strict* if it uses its argument at least once. The strictness of a function from A to B can be enforced by the type $(A \otimes !A) \multimap B$.

1. Show how to represent a strict function $\lambda^s x{:}A.\ M$ under this encoding.
2. Show how to represent an application of a strict function M to an argument N.
3. Give natural evaluation rules for strict functions and strict applications.
4. Show the corresponding computation under the encoding of strict functions in the linear λ-calculus.
5. Discuss the merits and difficulties of the given encoding of the strict in the linear λ-calculus.

Exercise 6.20 In the exercise we explore the *affine* λ-calculus. In an affine hypothetical judgment, each assumption can be used at most once. Therefore, it is like linear logic except that affine hypotheses need not be used.

The affine hypothetical judgment $\Gamma; \Delta \vdash^a A\ \mathit{true}$ is characterized by the hypothesis rule

$$\overline{\Gamma; \Delta, x{:}A\ \mathit{true} \vdash^a A\ \mathit{true}}$$

and the substitution principle

if $\Gamma; \Delta \vdash^a A\ \mathit{true}$ *and* $\Gamma; \Delta', A\ \mathit{true} \vdash^a C\ \mathit{true}$ *then* $\Gamma; \Delta, \Delta' \vdash^a C\ \mathit{true}$.

1. State the remaining hypothesis rule and substitution principle for unrestricted hypotheses.
2. Give introduction and elimination rules for affine implication ($A \rightsquigarrow B$) simultaneous conjunction, alternative conjunction, and truth. Note that there is only one form of truth: since assumptions need not be used, the multiplicative and additive form coincide.
3. Give a proof term assignment for affine logic.
4. We can map affine logic to linear logic by translating every affine function $A \rightsquigarrow B$ to a linear function $(A \& 1) \multimap B$. Give a corresponding translation for all proof terms from the affine logic to linear logic.
5. We can also map affine logic to linear logic by translating every affine function $A \rightsquigarrow B$ into function $A \multimap (B \otimes \top)$. Again give a corresponding translation for all proof terms from affine logic to linear logic.
6. Discuss the relative merits of the two translations.
7. [Extra Credit] Carefully formulate and prove the correctness of one of the two translations.

Chapter 7

Linear Type Theory

The distinction between logic and type theory is not always clear in the literature. From the judgmental point of view, the principal judgments of logic are *A is a proposition* (*A prop*) and *A is true* (*A true*). This may be different for richer logics. For example, in temporal logic we may have a basic judgment *A is true at time t*. However, it appears to be a characteristic that the judgments of logic are concerned with the study or propositions and truth.

In type theory, we elevate the concept of *proof* to a primary concept. In constructive logic this is important because proofs give rise to computation under reduction, as discussed in the chapter on linear functional programming (Chapter 6). Therefore, our primary judgment has the form *M is a proof of A* ($M : A$). This has the alternative interpretation *M is an object of type A*. So the principal feature that distinguishes a type theory from a logic is the internal notion of proof. But proofs are programs, so right from the outset, type theory has an internal notion of program and computation which is lacking from logic.

The desire to internalize proofs and computation opens a rich design space for the judgments defining type theory. We will not survey the different possibilities, but we may map out a particular path that is appropriate for linear type theory. We may occasionally mention alternative approaches or possibilities.

7.1 Dependent Types

The foundation of linear type theory was already laid in Section 6.1 where we introduced a *propositional linear type theory* through the notion of proof terms. We call it propositional, because the corresponding logic does not allow quantification.

$$
\begin{array}{llrll}
\text{Propositions} & A & ::= & P & \text{Atoms} \\
 & & & \mid A_1 \multimap A_2 \mid A_1 \otimes A_2 \mid \mathbf{1} & \text{Multiplicatives} \\
 & & & \mid A_1 \& A_2 \mid \top \mid A_1 \oplus A_2 \mid \mathbf{0} & \text{Additives} \\
 & & & \mid A \supset B \mid {!A} & \text{Exponentials}
\end{array}
$$

We now reconsider the quantifiers, $\forall x.\ A$ and $\exists x.\ A$. In the first-order linear logic we developed, the quantifiers range over a single (unspecified) domain. We may thus think of first-order logic as the study of quantification independently of any particular domain. This is accomplished by not making any assumptions about the domain of quantification. In contrast, first-order arithmetic arises if we introduce natural numbers and allow quantifiers to range specifically over natural numbers. This suggests to generalize the quantifiers to $\forall x{:}\tau.\ A$ and $\exists x{:}\tau.\ A$, where τ is a type.

In type theory, we may identify types with propositions. Therefore, we may label a quantifier with A instead of inventing a new syntactic category τ of types. Data types, such as the natural numbers, then have to be introduced as new types A together with their introduction and elimination rules. We postpone the discussion of numbers and other data types to Section **??**. Here, we are most interested in understanding the nature of the quantifiers themselves once they are typed.

Universal Quantification. Before, the introduction rule for $\forall x.\ A$ required us to prove $[a/x]A$ for a new parameter a. This stays essentially the same, except that we are allowed to make a typing assumption for a.

$$\frac{\Gamma, a{:}B; \Delta \vdash [a/x]A\ \mathit{true}}{\Gamma; \Delta \vdash \forall x{:}B.\ A\ \mathit{true}}\ \forall\mathrm{I}^a$$

Note that the parameter a is treated in an unrestricted manner. In other words, the object a we assume to exist is *persistent*, it is independent of the state. This avoids the unpleasant situation where a proposition C may talk about an object that no longer exists in the current state as defined via the linear hypotheses. See Exercise **??** for an exploration of this issue.

The rule above is written as part of a logic, and not as part of a type theory. We can obtain the corresponding type-theoretical formulation by adding proof terms. In this case, the proof of the premise is a function that maps an object N of type B to a proof of $[N/x]A$. We write this function as an ordinary λ-abstraction. We will also now use the same name x for the parameter and the bound variable, to remain closer to the traditions functional programming and type theory.

$$\frac{\Gamma, x{:}B; \Delta \vdash M : A}{\Gamma; \Delta \vdash \lambda x{:}B.\ M : \forall x{:}B.\ A}\ \forall\mathrm{I}$$

It is implicit here that x must be new, that is, it may not already be declared in Γ or Δ. This can always be achieved via renaming of the bound variable x in the proof term and the type of the conclusion. Note that the variable x may occur in A. This represents a significant change from the propositional case, where the variables in Γ and Δ occur only in proof terms, but not propositions.

In the corresponding elimination rule we now need to check that the term we use to instantiate the universal quantifier has the correct type. The proof term for the result is simply application.

$$\frac{\Gamma; \Delta \vdash M : \forall x{:}B.\ A \qquad \Gamma; \cdot \vdash N : B}{\Gamma; \Delta \vdash M\,N : [N/x]A}\ \forall E$$

Because $x{:}B$ may be used in an unrestricted manner in the proof of A, the proof of B may not depend on any linear hypotheses.

The local reduction is simply β-reduction, the local expansion is η-expansion. We write these out on the proof terms.

$$\begin{array}{rcl} (\lambda x{:}B.\ M)\,N & \longrightarrow_\beta & [N/x]M \\ M : \forall x{:}B.\ A & \longrightarrow_\eta & \lambda x{:}B.\ M\,x \end{array}$$

When viewed as a type, the universal quantifier is a *dependent function type*. The word "dependent" refers to the fact that the type of the the result of the function $M : \forall x{:}B.\ A$ *depends* on the actual argument N since it is $[N/x]A$. In type theory this is most often written as $\Pi x{:}B.\ A$ instead of $\forall x{:}B.\ A$. The dependent function type is a generalization of the ordinary function type (see Exercise **??**) and in type theory we usually view $B \supset A$ as an abbreviation for $\forall x{:}B.\ A$ for some x that does not occur free in A. This preserves the property that there is exactly one rule for each form of term. In this section we will also use the form $B \rightarrow A$ instead of $B \supset A$ to emphasize the reading of the propositions as types.

As a first simple example, consider a proof of $\forall x{:}i.\ P(x) \multimap P(x)$ for some arbitrary type i and predicate P on objects of type i.

$$\dfrac{\dfrac{\dfrac{}{x{:}i; u{:}P(x) \vdash u : P(x)}\ u}{x{:}i; \cdot \vdash \hat{\lambda} u{:}P(x).\ u : P(x) \multimap P(x)}\ {\multimap} I}{\cdot; \cdot \vdash \lambda x{:}i.\ \hat{\lambda} u{:}P(x).\ u : \forall x{:}i.\ P(x) \multimap P(x)}\ \forall I$$

This proof is very much in the tradition of first-order logic—the added expressive power of a type theory is not exploited. We will see more examples later, after we have introduced the existential quantifier.

Existential Quantification. There is also a dependent version of existential quantification. For reasons similar to the existential quantifier, the witness N for the truth of $\exists x{:}A.\ B$ is persistent and cannot depend on linear hypotheses.

$$\frac{\Gamma; \cdot \vdash N : A \qquad \Gamma; \Delta \vdash M : [N/x]B}{\Gamma; \Delta \vdash N \mathbin{!\otimes^{\exists x.\ B}} M : \exists x{:}A.\ B}\ \exists I$$

Unfortunately, we need to annotate the new term constructor with most of its type in order to guarantee uniqueness. The problem is that even if we know N and $[N/x]B$, we cannot in general reconstruct B uniquely (see Exercise **??**). The notation $N \mathbin{!\otimes} M$ is a reminder that pairs of this form are exponential, not additive. In fact, $\exists x{:}A.\ B$ is a dependent generalization of the operator $A \mathbin{!\otimes} B$ defined either by introduction and eliminations or via notational definition as $(!A) \otimes B$. We have already seen the reverse, $A \mathbin{\otimes!} B$ in Section 5.5 and Exercise 5.2.

The corresponding elimination is just a dependent version of the ordinary existential elimination.

$$\frac{\Gamma; \Delta \vdash M : \exists x{:}A.\ B \qquad \Gamma, x{:}A; \Delta', u{:}B \vdash N : C}{\Gamma; (\Delta, \Delta') \vdash \mathsf{let}\, x \mathbin{!\otimes} u = M \,\mathsf{in}\, N : C}\ \exists\mathrm{E}$$

Here, x and u must be new with respect to Γ and Δ'. In particular, they cannot occur in C. Also note that x is unrestricted while u is linear, as expected from the introduction rule.

Again, we write out the local reductions on proof terms.

$$\begin{array}{rcl}
\mathsf{let}\, x \mathbin{!\otimes} u = M_1 \mathbin{!\otimes} M_2 \,\mathsf{in}\, N & \longrightarrow_\beta & [M_1/x][M_2/u]N \\
M : A \mathbin{!\otimes} B & \longrightarrow_\eta & \mathsf{let}\, x \mathbin{!\otimes} u = M \,\mathsf{in}\, x \mathbin{!\otimes} u
\end{array}$$

Here we have omitted the superscript on the $!\otimes$ constructor for the sake of brevity.

Type Families. In order for a dependent type to be truly dependent, we must have occurrences of the quantified variables in the body of the type. In the example above we had $\forall x{:}i.\ P(x) \multimap P(x)$. But what is the nature of P? Logically speaking, it is a predicate on objects of type i. Type-theoretically speaking, it is a *type family*. That is, $P(M)$ is a type for every object M of type i. In such a type we call M the *index object*. It should now be clear that well-formedness of types is not longer a trivial issue the way it has been so far. We therefore augment our collections of judgments with a new one, A *is a type* written as $A : \mathrm{type}$.

Since a type such as $P(x) \multimap P(x)$ may depend on some parameters, we consider a hypothetical judgment of the form

$$\Gamma; \cdot \vdash A : \mathrm{type}.$$

There are no linear hypotheses because it not clear what such a judgment would mean, as hinted above. One possible answer has been given by Ishtiaq and Pym [IP98], but we restrict ourselves here to the simpler case.

Most rules for this judgment are entirely straightforward; we show only three representative ones.

$$\frac{\Gamma; \cdot \vdash A : \mathrm{type} \qquad \Gamma; \cdot \vdash B : \mathrm{type}}{\Gamma; \cdot \vdash A \multimap B : \mathrm{type}} \multimap\mathrm{F}$$

$$\frac{\Gamma; \cdot \vdash A : \mathrm{type} \qquad \Gamma, x{:}A; \cdot \vdash B : \mathrm{type}}{\Gamma; \cdot \vdash \forall x{:}A.\ B : \mathrm{type}} \forall\mathrm{F}$$

$$\frac{\Gamma; \cdot \vdash A : \mathrm{type} \qquad \Gamma, x{:}A; \cdot \vdash B : \mathrm{type}}{\Gamma; \cdot \vdash \exists x{:}A.\ B : \mathrm{type}} \exists\mathrm{F}$$

Atomic types $a\ M_1 \ldots M_n$ consist of a type family a indexed by objects $M_1, \ldots, M_n$. We introduce each such family with a separate formation rule. For example,

$$\frac{\Gamma; \cdot \vdash M : i}{\Gamma; \cdot \vdash P\ M : \mathrm{type}} P\mathrm{F}$$

would be the formation rule for the type family P considered above. Later it will be convenient to collect this information in a *signature* instead (see Section **??**).

In the next section we will see some examples of type families from functional programming.

7.2 Dependently Typed Data Structures

One potentially important application of dependent types lies functional programming. Here we can use certain forms of dependent types to capture data structure invariants concisely. As we will see, there are also some obstacles to the practical use of such type systems.

As an example we will use lists with elements of some type A, indexed by their length. This is of practical interest because dependent types may allow us to eliminate bounds checks statically, as demonstrated in [XP98].

Natural Numbers. First, the formation and introduction rules for natural numbers in unary form.

$$\frac{}{\Gamma; \cdot \vdash \mathsf{nat} : \mathrm{type}} \mathsf{nat}\mathrm{F}$$

$$\frac{}{\Gamma; \cdot \vdash 0 : \mathsf{nat}} \mathsf{nat}\mathrm{I}_0 \qquad \frac{\Gamma; \Delta \vdash M : \mathsf{nat}}{\Gamma; \Delta \vdash \mathsf{s}(M) : \mathsf{nat}} \mathsf{nat}\mathrm{I}_1$$

There a two destructors: one a case construct and one operator for iteration. We give these in the schematic form, as new syntax constructors, and as constants.

Schematically, f is defined by cases if it satisfies

$$\begin{array}{rcl} f(0) & = & N_0 \\ f(\mathsf{s}(n)) & = & N_1(n) \end{array}$$

Here, $N_1(n)$ indicates that the object N_1 may depend on n. However, this form does not express linearity conditions. If we write it as a new language constructor, we have (avoiding excessive syntactic sugar)

$$\mathsf{case}^{\mathsf{nat}}(M; N_0, n.\ N_1)$$

with the rule

$$\frac{\Gamma; \Delta \vdash M : \mathsf{nat} \qquad \Gamma; \Delta' \vdash N_0 : C \qquad \Gamma; \Delta', n : \mathsf{nat} \vdash N_1 : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{case}^{\mathsf{nat}}(M; N_0, n.\ N_1) : C}\ \mathsf{nat}\mathrm{E}_{\mathrm{case}}$$

Note that the elimination is additive in character, since exactly one branch of the case will be taken. The local reductions and expansion are simple:

$$\begin{array}{rcl} \mathsf{case}^{\mathsf{nat}}(0; N_0, n.\ N_1) & \longrightarrow_L & N_0 \\ \mathsf{case}^{\mathsf{nat}}(\mathsf{s}(M); N_0, n.\ N_1) & \longrightarrow_L & [M/n]N_1 \\ M : \mathsf{nat} & \longrightarrow_\eta & \mathsf{case}^{\mathsf{nat}}(M; 0, n.\ \mathsf{s}(n)) \end{array}$$

We could also introduce $\mathsf{case}^{\mathsf{nat}}$ as a constant with linear typing. This would violate our orthogonality principle. However, for any given type C we can define a “canonical” higher-order function implementing a case-like construct with the expected operational behavior.

$$\begin{array}{rcl} \mathsf{casenat}_C & : & \mathsf{nat} \multimap (C \mathbin{\&} (\mathsf{nat} \multimap C)) \multimap C \\ & = & \hat{\lambda} n{:}\mathsf{nat}.\ \hat{\lambda} c{:}(C \mathbin{\&} (\mathsf{nat} \multimap C)).\ \mathsf{case}^{\mathsf{nat}}(n; \mathsf{fst}\ c, n.\ \mathsf{snd}\ c \mathbin{\hat{}} n) \end{array}$$

Schematically, f is defined by iteration if it satisfies

$$\begin{array}{rcl} f(0) & = & N_0 \\ f(\mathsf{s}(n)) & = & N_1(f(n)) \end{array}$$

Here, N_1 can refer to the value of f on the predecessor, but not to n itself. As a language constructor:

$$\mathsf{it}^{\mathsf{nat}}(M; N_0, r.\ N_1)$$

with the rule

$$\frac{\Gamma; \Delta \vdash M : \mathsf{nat} \qquad \Gamma; \cdot \vdash N_0 : C \qquad \Gamma; r{:}C \vdash N_1 : C}{\Gamma; \Delta \vdash \mathsf{it}^{\mathsf{nat}}(M, N_0, r.\ N_1) : C}\ \mathsf{nat}\mathrm{E}_{\mathrm{it}}$$

Note that N_1 will be used as many times as M indicates, and can therefore not depend on any linear variables. We therefore also do not allow the branch for 0 to depend on linear variables. In this special data type this would be possible, since each natural number contains exactly one base case (see Exercise **??**).

$$\begin{array}{rcl}\mathsf{it}^{\mathsf{nat}}(0; N_0, r.\, N_1) & \longrightarrow_L & N_0 \\ \mathsf{it}^{\mathsf{nat}}(\mathsf{s}(M); N_0, r.\, N_1) & \longrightarrow_L & [\mathsf{it}^{\mathsf{nat}}(M; N_0, r.\, N_1)/r]N_1 \\ M : \mathsf{nat} & \longrightarrow_\eta & \mathsf{it}^{\mathsf{nat}}(M; 0, r.\, s(r))\end{array}$$

From this we can define a canonical higher-order function for each result type C implementing iteration.

$$\begin{array}{rcl}\mathsf{itnat}_C & : & \mathsf{nat} \multimap (C \,\&\, (C \multimap C)) \to C \\ & = & \hat{\lambda} n{:}\mathsf{nat}.\ \lambda c{:}C \,\&\, (C \multimap C).\ \mathsf{it}^{\mathsf{nat}}(n; \mathsf{fst}\, c, r.\ \mathsf{snd}\, c \,\hat{}\, r)\end{array}$$

More interesting is to define an operator for primitive recursion. Note the linear typing, which requires that at each stage of iteration we either use the result from the recursive call or the predecessor, but not both.

$$\mathsf{recnat}_C \quad : \mathsf{nat} \multimap (C \,\&\, ((\mathsf{nat} \,\&\, C) \multimap C)) \to C$$

The definition of this recursor is subject of Exercise **??**.

Lists. Next we define the data type of lists. We leave the type of elements open, that is, list is a *type constructor*. In addition, lists are indexed by their length. We therefore have the following formation rule

$$\frac{\Gamma; \cdot \vdash A : \mathrm{type} \qquad \Gamma; \cdot \vdash M : \mathsf{nat}}{\Gamma; \cdot \vdash \mathsf{list}_A(M) : \mathrm{type}}\ \mathsf{listF}$$

There are two introduction rules for lists: one for the empty list (nil) and one for a list constructor (cons). Note that we have to take care to construct the proper index objects. In order to simplify type-checking and the description of the operational semantics, we make the length of the list explicit as an argument to the constructor. In practice, this argument can often be inferred and in many cases does not need to be carried when a program is executed. We indicate this informally by writing this dependent argument as a subscript.

$$\frac{}{\Gamma; \cdot \vdash \mathsf{nil}^A : \mathsf{list}_A(0)}\ \mathsf{list}I_0$$

$$\frac{\Gamma; \cdot \vdash N : \mathsf{nat} \qquad \Gamma; \Delta \vdash H : A \qquad \Gamma; \Delta' \vdash L : \mathsf{list}_A(N)}{\Gamma; \Delta, \Delta' \vdash \mathsf{cons}_N\ H\ L : \mathsf{list}_A(\mathsf{s}(N))}\ \mathsf{list}I_1$$

Again, there are two elimination constructs: one for cases and one for iteration. A function for primitive recursion can be defined.

Schematically, f is defined by cases over a list l if it satisfies:

$$\begin{array}{rcl}f(\mathsf{nil}) & = & N_0 \\ f(\mathsf{cons}_N\, H\, L) & = & N_1(N, H, L)\end{array}$$

Here we supplied an additional argument to N_1 since we cannot statically predict the length of the list L. This also complicates the typing rule for the case construct, in addition to the linearity conditions.

$$\mathsf{case}^{\mathsf{list}}(M; N_0, n.\ h.\ l.\ N_1)$$

with the rule

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash M : \mathsf{list}_A(N) \\ \Gamma; \Delta' \vdash N_0 : C(0) \\ \Gamma, n{:}\mathsf{nat}; \Delta', h{:}A, l{:}\mathsf{list}_A(n) \vdash N_1 : C(\mathsf{s}(n))\end{array}}{\Gamma; \Delta, \Delta' \vdash \mathsf{it}^{\mathsf{list}}(M; N_0, n.\ h.\ l.\ N_1) : C(N)}\ \mathsf{nat}\mathrm{E}_{\mathsf{case}}$$

The novelty in this rule is that C is normally just a type; here it is a type family indexed by a natural number. This is necessary so that, for example

$$\begin{array}{lcl} id_A & : & \forall n{:}\mathsf{nat}.\ \mathsf{list}_A(n) \multimap \mathsf{list}_A(n) \\ & = & \lambda n{:}\mathsf{nat}.\ \lambda l{:}\mathsf{list}_A(n).\ \mathsf{case}^{\mathsf{list}}(M; 0, n.\ h.\ l'.\ \mathsf{cons}_n\, h\, l') \end{array}$$

type-checks. Note that here the first branch has type $\mathsf{list}_A(0)$ and the second branch has type $\mathsf{list}_A(\mathsf{s}(n))$, where n is the length of the list l' which stands for the tail of l. Local reduction and expansion are straightforward, given the intuition above.

$$\begin{array}{rcl} \mathsf{case}^{\mathsf{list}}(\mathsf{nil}; N_0; n.\ h.\ l.\ N_1) & \longrightarrow_L & N_0 \\ \mathsf{case}^{\mathsf{list}}(\mathsf{cons}_N\, H\, L; N_0; n.\ h.\ l.\ N_1) & \longrightarrow_L & [N/n, H/h, L/l]N_1 \\ M : \mathsf{list}_A(N) & \longrightarrow_\eta & \mathsf{case}^{\mathsf{list}}(M; \mathsf{nil}; n.\ h.\ l.\ \mathsf{cons}_n\, h\, l) \end{array}$$

If we write a higher-order case function it would have type

$$\mathsf{caselist}_{A,C} : \forall m{:}\mathsf{nat}.\ \mathsf{list}_A(m) \multimap (C(0) \,\&\, (\forall n{:}\mathsf{nat}.\ A \otimes \mathsf{list}_A(n) \multimap C(\mathsf{s}(n)))) \multimap C(m)$$

The iteration constructs for lists follows a similar idea. Schematically, f is defined by iteration over a list if it satisfies

$$\begin{array}{rcl} f(\mathsf{nil}) & = & N_0 \\ f(\mathsf{cons}_N\, H\, L) & = & N_1(N, H, f(L)) \end{array}$$

As a language constructor:

$$\mathsf{it}^{\mathsf{list}}(M; N_0, n.\ h.\ r.\ N_1)$$

with the rule

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash M : \mathsf{list}_A(N) \\ \Gamma; \cdot \vdash N_0 : C(0) \\ \Gamma, n{:}\mathsf{nat}; h{:}A, r{:}C(n) \vdash N_1 : C(\mathsf{s}(n))\end{array}}{\Gamma; \Delta \vdash \mathsf{it}^{\mathsf{list}}(M; N_0, n.\ h.\ r.\ N_1) : C(N)}\ \mathsf{nat}\mathrm{E}_{\mathsf{it}}$$

Local reduction and expansion present no new ideas.

$$\begin{array}{rcl}
\mathsf{it}^{\mathsf{list}}(\mathsf{nil}; N_0; n.\ h.\ r.\ N_1) & \longrightarrow_L & N_0 \\
\mathsf{it}^{\mathsf{list}}(\mathsf{cons}_N\ H\ L; N_0; n.\ h.\ r.\ N_1) & \longrightarrow_L & [N/n, H/h, \mathsf{it}^{\mathsf{list}}(L; N_0; n.\ h.\ r.\ N_1)/r]N_1 \\
M : \mathsf{list}_A(N) & \longrightarrow_\eta & \mathsf{it}^{\mathsf{list}}(M; \mathsf{nil}; n.\ h.\ r.\ \mathsf{cons}_n\ h\ r)
\end{array}$$

We can then define the following higher-order constant.

$$\mathsf{itlist}_{A,C} \quad : \quad \forall m{:}\mathsf{nat}.\ \mathsf{list}_A(m) \multimap (C(0) \& (\forall n{:}\mathsf{nat}.\ C(n) \multimap C(\mathsf{s}(n)))) \to C(m)$$

Note that, once again, C is a type family indexed by a natural number.

We now consider some functions on lists and their types. When we append two lists, the length of the resulting list will be the sum of the lengths of the two lists. To express this in our language, we first program addition.

$$\begin{array}{rcl}
\mathit{plus} & : & \mathsf{nat} \multimap \mathsf{nat} \multimap \mathsf{nat} \\
 & = & \hat{\lambda} x{:}\mathsf{nat}.\ \mathsf{it}^{\mathsf{nat}}(x; \hat{\lambda} y{:}\mathsf{nat}.\ y; r.\ \hat{\lambda} y{:}\mathsf{nat}.\ \mathsf{s}(r\,\hat{}\,y))
\end{array}$$

Note that the iteration is at type $\mathsf{nat} \multimap \mathsf{nat}$ so that $r : \mathsf{nat} \multimap \mathsf{nat}$ in the second branch.

$$\begin{array}{rcl}
\mathit{append}_A & : & \forall n{:}\mathsf{nat}.\ \forall m{:}\mathsf{nat}.\ \mathsf{list}_A(n) \multimap \mathsf{list}_A(m) \multimap \mathsf{list}_A(\mathit{plus}\ n\ m) \\
 & = & \lambda n{:}\mathsf{nat}.\ \lambda m{:}\mathsf{nat}.\ \hat{\lambda} l{:}\mathsf{list}_A(n). \\
 & & \quad \mathsf{it}^{\mathsf{list}}(l;\ \hat{\lambda} k{:}\mathsf{list}_A(m).\ k, \\
 & & \qquad\qquad p.\ h.\ r.\ \hat{\lambda} k{:}\mathsf{list}_A(m).\ \mathsf{cons}_p\ h\ (r\,\hat{}\,k))
\end{array}$$

This example illustrates an important property of dependent type systems: the type of the function append_A contains a defined function (in this case plus). This means we need to compute in the language in order to obtain the type of an expression. For example

$$\mathit{append}_A\ 0\ 0\ \mathsf{nil}_A\ \mathsf{nil}_A \quad : \mathsf{list}_A(\mathit{plus}\ 0\ 0)$$

by the rules for dependent types. But the result of evaluating this expression will be nil_A which has type $\mathsf{list}_A(0)$. Fortunately, $\mathit{plus}\ 0\ 0$ evaluates to 0. In general, however, we will not be able to obtain the type of an expression purely by computation, but we have to employ equality reasoning. This is because the arguments to a function will not be known at the time of type-checking. For example, to type-check the definition of append_A above, we obtain the type

$$\mathsf{list}_A(\mathsf{s}(\mathit{plus}\ p\ m))$$

for N_1 (the second branch) for two parameters p and m. The typing rules for $\mathsf{it}^{\mathsf{nat}}$ require this branch to have type

$$\mathsf{list}_A(\mathit{plus}\ (\mathsf{s}(p))\ m)$$

since the type family $C(n) = \mathsf{list}_A(\mathit{plus}\ n\ m)$ and the result of the second branch should have type $C(\mathsf{s}(n))$.

Fortunately, in this case, we can obtain the first type from the second essentially by some local reductions. In order for dependent type theories to be useful for functional programming we therefore need the rule of *type conversion*

$$\frac{\Gamma; \Delta \vdash M : A \qquad \Gamma; \cdot \vdash A = B : \text{type}}{\Gamma; \Delta \vdash M : B}\,\mathsf{conv}$$

where the $A = B$: type is a new judgment of *definitional equality*. At the very least the example above suggests that if M and N can be related by a sequence of reduction steps applied somewhere in M and N, then they should be considered equal. If this question is decidable is not at all clear and has to be reconsidered for each type theory in detail.

In an *extensional type theory*, such as the one underlying Nuprl [C$^+$86], we allow essentially arbitrarily complex reasoning (including induction) in order to prove that $A = B$: type. This means that conversion and also type-checking in general are undecidable—the judgment of the type theory are no longer analytic, but synthetic. This cast some doubt on the use of this type theory as a functional programming language.

In an *intensional type theory*, such as later type theories developed by Martin-Löf [ML80, NPS90, CNSvS94], definitional equality is kept weak and thereby decidable. The main judgment of the type theory remains analytic, which is desirable in the design of a programming language.

However, there is also a price to pay for a weak notion of equality. It means that sometimes we will be unable to type-check simple (and correct) functions, because the reason for their correctness requires inductive reasoning. A simple example might be

$$rev_A \quad : \quad \forall n{:}\mathsf{nat}.\ \mathsf{list}_A(n) \multimap \mathsf{list}_A(n)$$

which reverses the elements of a list. Depending on its precise formulation, we may need to know, for example, that

$$n{:}\mathsf{nat}, m{:}\mathsf{nat}; \cdot \vdash \mathsf{list}_A(plus\, n\, m) = \mathsf{list}_A(plus\, m\, n) : \text{type}$$

which will not be the case in an intensional type theory.

We circumvent this problem by introducing a new proposition (or type, depending on one's point of view) $eq\, N\, M$ for index objects N and M of type nat. Objects of this types are explicit proofs of equality for natural numbers and should admit induction (which is beyond the scope of these notes). The type of *rev* would then be rewritten as

$$rev_A \quad : \quad \forall n{:}\mathsf{nat}.\ \mathsf{list}_A(n) \multimap \exists m{:}\mathsf{nat}.\ \exists p{:}eq\, n\, m.\ \mathsf{list}_A(m)$$

In other words, in order to have decidable type-checking, we sometimes need to make correctness proofs explicit. This is not surprising, given the experience that correctness proofs can often be difficult.

In practical languages such as ML, it is therefore difficult to include dependent types. The approach taken in [Xi98] is to restrict index objects to be drawn from a domain with a decidable equality theory. This appears to be a reasonable compromise that can make the expressive power of dependent types available to the programmer without sacrificing decidable and efficient type-checking.

7.3 Logical Frameworks

In the previous section, we illustrated how (linear) type theory could be used as the foundation of (linear) functional programming. The demands of a complete functional language and, in particular, the presence of data types and recursion makes it difficult to attain decidable type-checking. In this section we discuss another application of linear type theory as the foundation for a *logical framework*.

A logical framework is a meta-language for the specification and implementation of deductive systems. This includes applications in various logics and programming languages For surveys of logical frameworks and their applications, see [Pfe96, BM01, Pfe01].

One of the most expressive current frameworks is LF [HHP93] which is based on a λ-calculus with dependent types. There are many elegant encodings of deductive systems in such a framework that can be found in the references above. However, there are also many systems occurring both in logic and programming languages, for which encodings are awkward. Examples are substructural logic (such as linear logic), calculi for concurrency (such as the π-calculus), or programming languages with state (such as Mini-ML with mutable references). Even for pure languages such as Haskell, some aspects of the operational semantics such as laziness and memoization are difficult to handle. This is because at a lower level of abstraction, the implementations of even the purest languages are essentially state-based.

In response to these shortcomings, a linear logical framework (LLF) has been developed [CP96, Cer96, CP98]. This framework solved some of the problems mentioned above. In particular, it allows natural encodings of systems with state. However, it did not succeed with respect to intrinsic notions of concurrency. We will try to give an intuitive explanation of why this is so after showing what the system looks like.

First, in a nutshell, the system LF. The syntax has the following form:

$$
\begin{array}{lcll}
\text{Types} & A & ::= & a\,M_1 \ldots M_n \mid \Pi x{:}A_1 \ldots A_2 \\
\text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\, M \mid M_1\, M_2
\end{array}
$$

Here, a ranges over *type families* indexed by object M_i, c ranger over *object constants*. Unlike the functional language above, these are not declared by formation rules, introduction rules, and elimination rules. In fact, we must avoid additional introduction and elimination rules because the corresponding local reductions make the equational theory (and therefore type checking) difficult to manage.

Instead, all judgments will be *parametric* in these constants! They are declared in a *signature* Σ which is a global analogue of the context Γ. We need kinds in order to classify type families.

$$\begin{array}{rrcl} \text{Kinds} & K & ::= & \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ \text{type} \\ \text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \end{array}$$

As a simple example, we consider a small imperative language. We distinguish expressions (which have a value, but no effect) from commands (which have an effect, but no value). Expressions are left unspecified, except that variables are allowed as expressions and that we must have boolean values true and false. In other words, our example is parametric in the language of expressions; the only requirements are that they cannot have an effect, and that they must allow variables. Commands are no-ops (skip), sequential composition $(C_1; C_2)$, parallel composition $(C_1 \parallel C_2)$, assignment $(x := e)$ and the allocation of a new local variable ($\mathbf{new}\, x.\ C$). We also have a conditional construct ($\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$) and a loop $\mathbf{loop}\ l.\ C$.

$$\begin{array}{rrcl} \text{Expression Types} & \tau & ::= & \mathbf{bool} \mid \ldots \\ \text{Expressions} & e & ::= & x \mid \mathbf{true} \mid \mathbf{false} \mid \ldots \\ \text{Commands} & C & ::= & \mathbf{skip} \mid C_1; C_2 \mid C_1 \parallel C_2 \mid x := e \mid \mathbf{new}\, x{:}\tau.\ C \\ & & & \mid \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \mid \mathbf{loop}\ l.\ C \end{array}$$

For the loop construct $\mathbf{loop}\ l.\ C$, we introduce a label l with scope C. This label is considered a new command in C, where invoking l corresponds to a copy of the loop body. Thus executing $\mathbf{loop}\ l.\ C$ reduces to executing $[\mathbf{loop}\ l.\ C/l]C$. Note that this only allows backwards jumps and does not model a general **goto**. Its semantics is also different from goto if it is not the "last" command in a loop (see Exercise **??** where you are also asked to model a **while**-loop using **loop**). The simplest infinite loop is $\mathbf{loop}\ l.\ l$.

We now turn to the representation of syntax. We have a framework type tp representing object language types, and index expressions by their object language type. We show the representation function $\ulcorner\tau\urcorner$ and $\ulcorner e\urcorner$.

$$\begin{array}{lcl@{\qquad}lcl} & & & \mathsf{tp} & : & \text{type} \\ \ulcorner\mathbf{bool}\urcorner & = & \mathsf{bool} & \mathsf{bool} & : & \mathsf{tp} \\[1ex] & & & \mathsf{var} & : & \mathsf{tp} \to \text{type} \\ & & & \mathsf{exp} & : & \mathsf{tp} \to \text{type} \\ \ulcorner x\urcorner & = & \mathsf{v}\,\ulcorner\tau\urcorner\,x & \mathsf{v} & : & \forall t{:}\mathsf{tp}.\ \mathsf{var}(t) \to \mathsf{exp}(t) \\ \ulcorner\mathbf{true}\urcorner & = & \mathsf{true} & \mathsf{true} & : & \mathsf{exp}(\mathsf{bool}) \\ \ulcorner\mathbf{false}\urcorner & = & \mathsf{false} & \mathsf{false} & : & \mathsf{exp}(\mathsf{bool}) \end{array}$$

One of the main points to note here is that variables of our imperative language are represented by variables in the framework. For the sake of convenience, we choose the same name for a variable and its representation. The type of such a variable in the framework will be $\mathsf{var}(\ulcorner\tau\urcorner)$ depending on its declared type. v is the coercion from a variable of type τ to an expression of type τ. It has to be indexed by a type t, because it could be applied to an expression of any type.

Commands on the other hand do not return a value, so their type is not indexed. In order to maintain the idea the variables are represented by variables, binding constructs in the object language (namely **new** and **loop**) must be translated so they bind the corresponding variable in the meta-language. This idea is called *higher-order abstract syntax* [PE88], since the type of such constructs in generally of order 2 (the constructors new and loop take functions as arguments).

$$\begin{array}{lcl}
\ulcorner \mathbf{skip} \urcorner & = & \mathsf{skip} \\
\ulcorner C_1 ; C_2 \urcorner & = & \mathsf{seq}\, \ulcorner C_1 \urcorner \ulcorner C_2 \urcorner \\
\ulcorner C_1 \parallel C_2 \urcorner & = & \mathsf{par}\, \ulcorner C_1 \urcorner \ulcorner C_2 \urcorner \\
\ulcorner x := e \urcorner & = & \mathsf{assign}\, \ulcorner \tau \urcorner\; x\; \ulcorner e \urcorner \quad \text{for } x \text{ and } e \text{ of type } \tau \\
\ulcorner \mathbf{new}\, x{:}\tau.\, C \urcorner & = & \mathsf{new}\, \ulcorner \tau \urcorner\, (\lambda x{:}\mathsf{var}(\ulcorner \tau \urcorner).\, \ulcorner C \urcorner) \\
\ulcorner \mathbf{if}\; e\; \mathbf{then}\; C_1\; \mathbf{else}\; C_2 \urcorner & = & \mathsf{if}\, \ulcorner e \urcorner \ulcorner C_1 \urcorner \ulcorner C_2 \urcorner \quad \text{for } e \text{ of type } \mathbf{bool} \\
\ulcorner \mathbf{loop}\, l.\, C \urcorner & = & \mathsf{loop}\, (\lambda l{:}\mathsf{cmd}.\, \ulcorner C \urcorner)
\end{array}$$

In the declarations below, wee see that assign and new need to be indexed by their type, and that a conditional command branches depending on a boolean expression. In that way only well-typed commands can be represented—others will be rejected as ill-typed in the framework.

$$\begin{array}{lcl}
\mathsf{cmd} & : & \text{type} \\
\mathsf{skip} & : & \mathsf{cmd} \\
\mathsf{seq} & : & \mathsf{cmd} \to \mathsf{cmd} \to \mathsf{cmd} \\
\mathsf{par} & : & \mathsf{cmd} \to \mathsf{cmd} \to \mathsf{cmd} \\
\mathsf{assign} & : & \forall t{:}\mathsf{tp}.\; \mathsf{var}(t) \to \mathsf{exp}(t) \to \mathsf{cmd} \\
\mathsf{new} & : & \forall t{:}\mathsf{tp}.\; (\mathsf{var}(t) \to \mathsf{cmd}) \to \mathsf{cmd} \\
\mathsf{if} & : & \mathsf{exp}(\mathsf{bool}) \to \mathsf{cmd} \to \mathsf{cmd} \\
\mathsf{loop} & : & (\mathsf{cmd} \to \mathsf{cmd}) \to \mathsf{cmd}
\end{array}$$

So far, we have not needed linearity. Indeed, the declarations above are just a standard means of representing syntax in LF using the expressive power of dependent types for data representation.

Next we would like to represent the operational semantics in the style of encoding that we have used in this class before, beginning with expressions. We assume that for each variable x of type τ we have an affine assumption $x = v$ for a value v of type τ. In a context $x_1 = v_1, \ldots, x_n = v_n$ all variables x_i must be distinct. In judgmental notation, our judgment would be $\Psi \vdash^a e \hookrightarrow v$, where Ψ represents the store and $\vdash^a$ is an affine hypothetical judgment. In our fragment it would be represented by only three rules

$$\frac{}{\Psi, x = v \vdash^a x \hookrightarrow v}\;\mathsf{ev\_var}$$

$$\frac{}{\Psi \vdash^a \mathbf{true} \hookrightarrow \mathbf{true}}\;\mathsf{ev\_true} \qquad \frac{}{\Psi \vdash^a \mathbf{false} \hookrightarrow \mathbf{false}}\;\mathsf{ev\_false}$$

Since our framework is linear and not affine, we need to take care of eliminating unconsumed hypotheses. We have

$$\begin{array}{rcl} \ulcorner x = v \urcorner & = & \mathsf{value}\,\ulcorner \tau \urcorner\, x \,\ulcorner v \urcorner \\ \ulcorner e \hookrightarrow v \urcorner & = & \mathsf{eval}\,\ulcorner \tau \urcorner \ulcorner e \urcorner \ulcorner t \urcorner \end{array}$$

Note that these will be represented as *type families* since judgments are represented as types and deductions as objects.

$$\begin{array}{lcl} \mathsf{value} & : & \forall t{:}\mathsf{tp}.\ \mathsf{var}(t) \to \mathsf{exp}(t) \to \mathrm{type} \\ \mathsf{eval} & : & \forall t{:}\mathsf{tp}.\ \mathsf{exp}(t) \to \mathsf{exp}(t) \to \mathrm{type} \end{array}$$

Note that value and eval are both type families with dependent kinds, that is, the type of the second and third index objects depends on the first index object t in both cases. In the fragment we consider here, the evaluation rules are straightforward, although we have to take care to consume extraneous resources in the case of variable.

$$\begin{array}{lcl} \mathsf{ev\_true} & : & \top \multimap \mathsf{eval\ bool\ true\ true} \\ \mathsf{ev\_false} & : & \top \multimap \mathsf{eval\ bool\ false\ false} \\ \mathsf{ev\_var} & : & \forall t{:}\mathsf{tp}.\ \forall x{:}\mathsf{var}(t).\ \forall v{:}\mathsf{exp}(t).\ \forall e{:}\mathsf{exp}(t). \\ & & \top \multimap \mathsf{value}\, t\, x\, v \multimap \mathsf{eval}\, t\, (\mathsf{v}\, t\, x)\, v \end{array}$$

What did we need so far? We have used unrestricted implication and universal quantification, linear implication and additive truth. If we had a binary operator for expressions we would also need an additive conjunction so that the values of variables are accessible in both branches. Kinds and signatures change only to the extent that the types embedded in them change.

$$\begin{array}{rrcl} \text{Kinds} & K & ::= & \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\ \mathrm{type} \\ \text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid c{:}A \\ \text{Types} & A & ::= & a\, M_1 \ldots M_n \mid \Pi x{:}A_1 \ldots A_2 \\ & & & \mid A_1 \multimap A_2 \mid A_1 \& A_2 \mid \top \\ \text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\, M_2 \\ & & & \mid \hat{\lambda} x{:}A.\ M \mid M_1 \,\hat{}\, M_2 \\ & & & \mid \langle M_1, M_2 \rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \\ & & & \mid \langle\, \rangle \end{array}$$

Note that in this fragment, uniform deductions are complete. Moreover, every term has a canonical form, which is a β-normal, η-long form. Canonical forms are defined as in Section 2.6 where the coercion from atomic to normal derivations is restricted to atomic propositions. This is important, because it allows a relatively simple algorithm for testing equality between objects, which is necessary because of the rule of type conversion in the type theory.

The appropriate notion of definitional equality here is then defined by the rules for β-reduction and η-expansion.

$$\frac{\Gamma, x{:}A; \Delta \vdash M : B \qquad \Gamma; \cdot \vdash N : A}{\Gamma; \Delta \vdash (\lambda x{:}A.\ M)\, N = [N/x]M : [N/x]B}$$

$$\frac{\Gamma; \Delta, u{:}A \vdash M : B \qquad \Gamma; \Delta' \vdash N : A}{\Gamma; \Delta, \Delta' \vdash (\hat{\lambda} u{:}A.\ M)\,\hat{}\,N = [N/u]M : B}$$

$$\frac{\Gamma; \Delta \vdash M_1 : A \qquad \Gamma; \Delta \vdash M_2 : B}{\Gamma; \Delta \vdash \mathsf{fst}\,\langle M_1, M_2\rangle = M_1 : A} \qquad \frac{\Gamma; \Delta \vdash M_1 : A \qquad \Gamma; \Delta \vdash M_2 : B}{\Gamma; \Delta \vdash \mathsf{snd}\,\langle M_1, M_2\rangle = M_2 : B}$$

$$\frac{\Gamma; \Delta \vdash M : \forall x{:}A.\ B}{\Gamma; \Delta \vdash M = \lambda x{:}A.\ M\, x : \forall x{:}A.\ B}$$

$$\frac{\Gamma; \Delta \vdash M : A \multimap B}{\Gamma; \Delta \vdash M = \hat{\lambda} u{:}A.\ M\,\hat{}\,u : A \multimap B}$$

$$\frac{\Gamma; \Delta \vdash M : A \& B}{\Gamma; \Delta \vdash M = \langle \mathsf{fst}\, M, \mathsf{snd}\, M\rangle : A \& B}$$

$$\frac{\Gamma; \Delta \vdash M : \top}{\Gamma; \Delta \vdash M = \langle\,\rangle : \top}$$

The remaining rules are reflexivity, symmetry, transitivity, and congruence rules for each constructors. It is by no means trivial that this kind of equality is decidable (see [CP98, VC00]). We would like to emphasize once again that in a dependent type theory, decidability of judgmental (definitional) equality is necessary to obtain a decidable type-checking problem.

Next we come to the specification of the operational semantics of commands. For this we give a natural specification in terms of two type families, $\mathsf{exec}\, D\, C$ where D is a label for the command, and $\mathsf{done}\, D$ which indicates that the command labeled D has finished. The labels here are not targets for jumps since they are not part of the program (see Exercise **??**). Note that there are no constructors for labels—we will generate them as parameters during the execution.

$$\begin{array}{rcl} \mathsf{lab} & : & \mathrm{type} \\ \mathsf{exec} & : & \mathsf{lab} \to \mathsf{cmd} \to \mathrm{type} \\ \mathsf{done} & : & \mathsf{lab} \to \mathrm{type} \end{array}$$

According to our general strategy, $\mathsf{exec}\ D\ C$ and $\mathsf{done}\ D$ will be part of our linear hypothesis. Without giving a structural operational semantics, we present the rules for execution of commands directly in the type theory.

skip returns immediately.

$$\mathsf{ex\_skip} \;:\; \forall D{:}\mathsf{lab}.\ \mathsf{exec}\ D\ \mathsf{skip} \multimap \mathsf{done}\ D$$

For sequential composition, we prevent execution of the second command until the first has finished.

$$\begin{array}{lll}\mathsf{ex\_seq} & : & \forall D{:}\mathsf{lab}.\ \forall C_1{:}\mathsf{cmd}.\ \forall C_2{:}\mathsf{cmd}.\\ & & \quad \mathsf{exec}\ D\ (\mathsf{seq}\ C_1\ C_2)\\ & & \quad \multimap (\exists d_1{:}\mathsf{lab}.\ \mathsf{exec}\ d_1\ C_1 \otimes (\mathsf{done}\ d_1 \multimap \mathsf{exec}\ D\ C_2))\end{array}$$

For parallel composition, both commands can proceed independently. The parallel composition is done, if both commands are finished. This is by no means the only choice of an operational semantics.

$$\begin{array}{lll}\mathsf{ex\_par} & : & \forall D{:}\mathsf{lab}.\ \forall C_1{:}\mathsf{cmd}.\ \forall C_2{:}\mathsf{cmd}.\\ & & \quad \mathsf{exec}\ D\ (\mathsf{par}\ C_1\ C_2)\\ & & \quad \multimap \exists d_1{:}\mathsf{lab}.\ \exists d_2{:}\mathsf{lab}.\ \ \mathsf{exec}\ d_1\ C_1 \otimes \mathsf{exec}\ d_2\ C_2\\ & & \qquad\qquad\qquad\qquad\qquad \otimes\ (\mathsf{done}\ d_1 \otimes \mathsf{done}\ d_2 \multimap \mathsf{done}\ D)\end{array}$$

For assignment, we need to consume the assumption $x = v'$ for the variable and assume the new value as $x = v$. We also want to allow assignment to an uninitialized variables, which is noted by an assumption $\mathsf{uninit}\ulcorner\tau\urcorner x$.

$$\begin{array}{lll}\mathsf{uninit} & : & \forall t{:}\mathsf{tp}.\ \mathsf{var}(t) \to \mathrm{type}\\ \mathsf{ex\_assign} & : & \forall D{:}\mathsf{lab}.\ \forall T{:}\mathsf{tp}.\ \forall X{:}\mathsf{var}(T).\ \forall E{:}\mathsf{exp}(T).\ \forall V{:}\mathsf{exp}(T).\\ & & \quad \mathsf{exec}\ D\ (\mathsf{assign}\ T\ X\ E)\\ & & \quad \multimap\ (\mathsf{eval}\ T\ E\ V \multimap \mathbf{0})\\ & & \qquad\quad \oplus\ \ ((\mathsf{uninit}\ T\ X \oplus \exists V'{:}\mathsf{exp}(T).\ \mathsf{value}\ T\ X\ V')\\ & & \qquad\qquad\quad \multimap \mathsf{value}\ T\ X\ V \otimes \mathsf{done}\ D)\end{array}$$

The new command creates a new, uninitialized variable.

$$\begin{array}{lll}\mathsf{ex\_new} & : & \forall D{:}\mathsf{lab}.\ \forall T{:}\mathsf{tp}.\ \forall C{:}\mathsf{var}(T) \to \mathsf{cmd}.\\ & & \quad \mathsf{exec}\ D\ (\mathsf{new}\ T\ (\lambda x{:}\mathsf{var}(T).\ C(x)))\\ & & \quad \multimap \exists y{:}\mathsf{var}(T).\ \mathsf{uninit}\ T\ y \otimes \mathsf{exec}\ D\ C(y)\end{array}$$

Note type of C, which depends on a variable x. In order to emphasize this point, we have created a variable y with a different name on the right-hand side. The $\mathsf{ex\_new}$ constant will be applied to a (framework) function $(\lambda z{:}\mathsf{var}(T).\ C')$ which is applied to x on the left-hand side and y on the right hand side. We then have

$$C(x) = (\lambda z{:}\mathsf{var}(T).\ C')\, x = [x/z]C'$$

where the second equality is a *definitionaly equality* which follows by β-conversion. On the right-hand side have instead

$$C(y) = (\lambda z{:}\mathsf{var}(T).\ C')\, y = [y/z]C',$$

again by β-conversion.

So here we take critical advantage of the rule of type conversion in order to construct our encoding.

We leave the straightforward rules for the conditional to Exercise **??**.

For loops, we give two alternative formulations. The first take advantage of definitional equality in order to perform substitution, thereby unrolling the loop.

$$\begin{array}{lll}\mathsf{ex\_loop} & : & \forall D{:}\mathsf{lab}.\ \forall C{:}\mathsf{cmd} \to \mathsf{cmd} \\ & & \quad \mathsf{exec}\ D\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l))) \\ & & \quad \multimap \mathsf{exec}\ D\ (C\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l))))\end{array}$$

Assume the command has form $\mathsf{loop}\, l.\ C'$. Then

$$\ulcorner \mathbf{loop}\, l.\ C' \urcorner = \mathsf{loop}\ (\lambda l{:}\mathsf{lab}.\ \ulcorner C' \urcorner).$$

Then the framework variable C will be instantiated with

$$C = \lambda l{:}\mathsf{cmd}.\ C'.$$

For the framework expression on the right-hand side we obtain

$$\begin{array}{l}(C\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l)))) \\ = (\lambda l{:}\mathsf{cmd}.\ C')\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l))) \\ = [(\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l)))/l]C' \\ = [\ulcorner \mathbf{loop}\, l.\ C' \urcorner/l]\ulcorner C' \urcorner \\ = \ulcorner [\mathbf{loop}\, l.\ C'/l]C' \urcorner\end{array}$$

Here the last equation follows by *compositionality*: substitution commutes with representation. This is a consequence of the decision to represent object-language variables by meta-language variables and can lead to very concise encodings (just as in this case). It means we do not have to explicitly axiomatize substitution, but we can let the framework take care of it for us. Since formalizing substitution can be a significant effort, this is a major advantage of higher-order abstract syntax over other representation techniques.

We can also avoid explicit substitution, instead adding an linear hypothesis that captures how to jump back to the beginning of a loop more directly.

$$\begin{array}{lll}\mathsf{ex\_loop}' & : & \forall D{:}\mathsf{lab}.\ \forall C{:}\mathsf{cmd} \to \mathsf{cmd} \\ & & \quad \mathsf{exec}\ D\ (\mathsf{loop}\ (\lambda l{:}\mathsf{cmd}.\ C(l))) \\ & & \quad \multimap \exists k{:}\mathsf{cmd}.\ \ (\forall d{:}\mathsf{lab}.\ \mathsf{exec}\ d\ k \multimap \mathsf{exec}\ d\ (C(k))) \\ & & \qquad\qquad\qquad \otimes\ \mathsf{exec}\ D\ (C(k))\end{array}$$

This completes our specification of the operational semantics, which is at a very high level of abstraction. Unfortunately, the concurrency in the specification requires leaving the fragment we have discussed so far and including multiplicative conjunction and existential quantification, at least. As mentioned before, there is a translation back into the uniform fragment that preserves provability. This translation, however, does not preserve a natural notion of equality on proofs, or a natural semantics in terms of logic program execution. It is a subject of current research to determine how concurrency can be introduced into the linear logical framework LLF in a way that preserves the right notion of equality.

So far, it seems we have not used the η-conversion. Our representation function is a bijection between syntactic categories of the object language and canonical forms of the representation type. Together with β-reduction, we need η-expansion to transform and arbitrary object to its canonical form. Without it we would have "exotic objects" that do not represent any expression in the language we are trying to model.

Bibliography

[ABCJ94] D. Albrecht, F. Bäuerle, J. N. Crossley, and J. S. Jeavons. Curry-Howard terms for linear logic. In ??, editor, *Logic Colloquium '94*, pages ??–?? ??, 1994.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[ACS98] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchrounous π-calculus. *Theoretical Computer Science*, 195(2):291–423, 1998.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[AP91] J.-M. Andreoli and R. Pareschi. Logic programming with sequent systems: A linear logic approach. In P. Schröder-Heister, editor, *Proceedings of Workshop to Extensions of Logic Programming, Tübingen, 1989*, pages 1–30. Springer-Verlag LNAI 475, 1991.

[AS01] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *Submitted*, 2001. A previous version presented at LICS'00.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

[Bib86] Wolfgang Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.

[Bie94] G. Bierman. On intuitionistic linear logic. Technical Report 346, University of Cambridge, Computer Laboratory, August 1994. Revised version of PhD thesis.

[BM01] David Basin and Seán Matthews. Logical frameworks. In Dov Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic*. Kluwer Academic Publishers, 2nd edition, 2001. In preparation.

[BS92] G. Bellin and P. J. Scott. On the π-calculus and linear logic. Manuscript, 1992.

[C$^+$86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Cer95] Iliano Cervesato. Petri nets and linear logic: a case study for logic programming. In M. Alpuente and M.I. Sessa, editors, *Proceedings of the Joint Conference on Declarative Programming (GULP-PRODE'95)*, pages 313–318, Marina di Vietri, Italy, September 1995. Palladio Press.

[Cer96] Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[CP98] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 1998. To appear in a special issue with invited papers from LICS'96, E. Clarke, editor.

[Doš93] Kosta Došen. A historical introduction to substructural logics. In Peter Schroeder-Heister and Kosta Došen, editors, *Substructural Logics*, pages 1–30. Clarendon Press, Oxford, England, 1993.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. Translated under the title *Investigations into Logical Deductions* in [Sza69].

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir93] J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[Her30] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Lettres de Varsovic*, 33, 1930.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[Hof00a] Martin Hofmann. Linear types and non-size increasing polynomial time computation. *Theoretical Computer Science*, 2000. To appear. A previous version was presented at LICS'99.

[Hof00b] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, November 2000. To appear. A previous version was presented as ESOP'00.

[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.

[HP97] James Harland and David Pym. Resource-distribution via boolean constraints. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, July 1997. Springer-Verlag LNAI 1249.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag LNCS 512.

[Hue76] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[IP98] Samin Ishtiaq and David Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[Kni89] Kevin Knight. Unification: A multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[Lin92] P. Lincoln. Linear logic. *ACM SIGACT Notices*, 23(2):29–37, Spring 1992.

[Mil92] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions of Logic Programming*, pages 242–265. Springer-Verlag LNCS 660, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML94] Per Martin-Löf. Analytic and synthetic judgements in type theory. In Paolo Parrini, editor, *Kant and Contemporary Epistemology*, pages 87–99. Kluwer Academic Publishers, 1994.

[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.

[MM76] Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Internal Report B76-16, Istituto di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MOM91] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *Journal on Foundations of Computer Science*, 2(4):297–399, December 1991.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[Pfe01] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 977–1061. Elsevier Science and MIT Press, 2001. In press.

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

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rob71] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[Sce93] A. Scedrov. A brief guide to linear logic. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 377–394. World Scientific Publishing Company, 1993. Also in Bulletin of the European Association for Theoretical Computer Science, volume 41, pages 154–165.

[SHD93] Peter Schroeder-Heister and Kosta Došen, editors. *Substructural Logics*. Number 2 in Studies in Logic and Computation. Clarendon Press, Oxford, England, 1993.

[Sta85] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. North-Holland Publishing Co., Amsterdam, 1969.

[Tai67] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal Of Symbolic Logic*, 32:198–212, 1967.

[Tro92] A. S. Troelstra. *Lectures on Linear Logic*. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.

[Tro93] A. S. Troelstra. Natural deduction for intuitionistic linear logic. Prepublication Series for Mathematical Logic and Foundations ML-93-09, Institute for Language, Logic and Computation, University of Amsterdam, 1993.

[VC00] Joseph C. Vanderwaart and Karl Crary. A simplified account of the metatheory of linear LF. Draft paper, September 2000.

[WW01] David Walker and Kevin Watkins. On linear types and regions. In *Proceedings of the International Conference on Functional Programming (ICFP'01)*. ACM Press, September 2001.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.