LOGICAL FRAMEWORKS—A BRIEF INTRODUCTION

FRANK PFENNING (fp+@cs.cmu.edu)
Carnegie Mellon University

Abstract. A logical framework is a meta-language for the formalization of deductive systems. We provide a brief introduction to logical frameworks and their methodology, concentrating on LF. We use first-order logic as the running example to illustrate the representations of syntax, natural deductions, and proof transformations.

We also sketch a recent formulation of LF centered on the notion of canonical form, and show how it affects proofs of adequacy of encodings.

Key words: Logical frameworks, type theory

1. Introduction

Deductive systems, given via axioms and rules of inference, are a common conceptual tool in mathematical logic and computer science. They are used to specify many varieties of logics and logical theories as well as aspects of programming languages such as type systems or operational semantics. A *logical framework* is a meta-language for the specification of deductive systems. A number of different frameworks have been proposed and implemented for a variety of purposes. In this brief introduction we highlight the major themes, concepts, and design choices for logical frameworks and provide pointers to the literature for further reading. We concentrate specifically on the LF type theory and we briefly mention other approaches below and in Section 5.

Logical frameworks are subject to the same general design principles as other specification and programming languages. They should be simple and uniform, providing concise means to express the concepts and methods of the intended application domains. Meaningless expressions should be detected statically and it should be possible to structure large specifications and verify that the components fit together. There are also concerns specific to logical frameworks. Perhaps most importantly, an implementation must

[0] This research was supported in part by the National Science Foundations under grant CCR-9988281.

be able to check deductions for validity with respect to the specification of a deductive system. Secondly, it should be feasible to prove (informally) that the representations of deductive systems in the framework are adequate so that we can trust formal derivations. We return to each of these points when we discuss different design choices for logical frameworks.

Historically, the first logical framework was Automath (de Bruijn, 1968) and its various languages, developed during the late sixties and early seventies. The goal of the Automath project was to provide a tool for the formalization of mathematics without foundational prejudice. Therefore, the logic underlying a particular mathematical development was an integral part of its formalization. Many of the ideas from the Automath language family have found their way into modern systems. The main experiment conducted within Automath was the formalization of Landau's *Foundations of Analysis*. In the early eighties the importance of constructive type theories for computer science was recognized through the pioneering work of Martin-Löf (Martin-Löf, 1980). On the one hand, this led to a number of systems for constructive mathematics and the extraction of functional programs from constructive proofs. On the other hand, it strongly influenced the design of LF (Harper et al., 1993), sometimes called the Edinburgh Logical Framework (ELF). Concurrent with the development of LF, frameworks based on higher-order logic and resolution were designed in the form of generic theorem provers (Paulson, 1986) and logic programming languages (Nadathur and Miller, 1988). The type-theoretic and logic programming approaches were later combined in the Elf language (Pfenning, 1991). At this point, there was a pause in the development of new frameworks, while the potential and limitations of existing systems were explored in numerous experiments (see Pfenning (1996)). The mid-nineties saw renewed activity with implementations of frameworks based on inductive definitions such as FS_0 (Feferman, 1988; Basin and Matthews, 1996) and ALF (Altenkirch et al., 1994), partial inductive definitions (Eriksson, 1994), substructural frameworks (Miller, 1994; Cervesato and Pfenning, 1996), rewriting logic (Martì-Oliet and Meseguer, 1993), and labelled deductive systems (Gabbay, 1994). A full discussion of these is beyond the scope of this introduction—the reader can find some remarks in handbook articles on the subject of logical frameworks (Basin and Matthews, 2001; Pfenning, 2001b).

Some researchers distinguish between logical frameworks and *meta-logical frameworks* (Basin and Constable, 1993), where the latter is intended as a meta-language for reasoning *about* deductive systems rather than *within* them. Clearly, any meta-logical framework must also provide means for specifying deductive systems, though with different goals. Space does

not permit a discussion of meta-logical frameworks in this survey.

The remainder of this introduction is organized as follows: in Section 2 we discuss the representation of the syntax of a logic and in Section 3 the representation of judgments and deductions. In Section 4 we provide a the details of a formulation of the dependently typed λ-calculus as a point of reference, before concluding in Section 5. As an example throughout we use natural deduction for first-order logic.

2. Abstract syntax

The specification of a deductive system usually proceeds in two stages: first we define the syntax of an object language and then the axioms and rules of inference. In order to concentrate on the meanings of expressions we ignore issues of concrete syntax and parsing and concentrate on specifying abstract syntax. Different framework implementations provide different means for customizing the parser in order to embed the desired object-language syntax.

As an example throughout we consider formulations of intuitionistic and classical first-order logic. In order to keep the length of this survey manageable, we restrict ourselves to the fragment containing implication, negation, and universal quantification. The reader is invited to test his or her understanding by extending the development to include a more complete set of connectives and quantifiers. Representations of first-order intuitionistic and classical logic in various logical frameworks can be found in the literature (see, for example, Harper et al. (1993), Pfenning (2001a)).

Our fragment of first-order logic is constructed from individual variables x, function symbols f, and predicate symbols p in the usual way. We assume each function and predicate symbol has a unique arity, indicated by a superscript, but generally omitted since it will be clear from the context. Individual constants are function symbols of arity 0 and propositional constants are predicate symbols of arity 0.

$$\begin{array}{lrcl} \text{Terms} & t & ::= & x \mid f^k(t_1, \ldots, t_k) \\ \text{Atoms} & P & ::= & p^k(t_1, \ldots, t_k) \\ \text{Formulas} & A & ::= & P \mid A_1 \supset A_2 \mid \neg A \mid \forall x.\, A \end{array}$$

We assume that there is an infinite number of variables x. The set of function and predicate symbols is left unspecified in the general development of logic. We therefore view our specification as open-ended. A commitment, say, to arithmetic would fix the available function and predicate symbols. We write x and y for variables, t and s for terms, and

A, B, and C for formulas. There are some important operations on terms and formulas required for the presentation of inference rules. Specifically, we need the notions of free and bound variable, the renaming of bound variables, and the operations of substitution $[t/x]s$ and $[t/x]A$, where the latter may need to rename variables bound in A in order to avoid variable capture. We assume that these operations are understood and do not define them formally. An assumption generally made in connection with variable names is the so-called *variable convention* (Barendregt, 1980) which states that expressions differing only in the names of their bound variables are considered identical. We examine to which extent various frameworks support this convention.

2.1. SIMPLY-TYPED REPRESENTATION

We would like to capture both the variable name convention and the validity of a framework object representing a term or formula *internally*. A standard method to achieve this is to introduce representation types. We begin with *simple types*. The idea is to introduce type constants i and o for object-level terms and formulas, respectively. Implication, for example, is then represented by a constant of type $\mathsf{o} \to (\mathsf{o} \to \mathsf{o})$, that is, a formula constructor taking two formulas as arguments employing the standard technique of Currying. We could now represent variables as strings or integers; instead, we use meta-language variables to model object-language variables. This requires that we enrich the representation language to include higher-order terms, which leads us to the simply-typed λ-calculus, $\lambda^{\to}$. As we will see from the adequacy theorem for our representation (Theorem 1), the methodology of logical frameworks only requires canonical forms, which are β-normal and η-long. We will capture this in the syntax of our representation language by allowing only β-normal forms; the fact that they are η-long is enforced in the typing rules (see Section 4).

$$\begin{array}{lll} \text{Types} & A & ::= a \mid A_1 \to A_2 \\ \text{Atomic Objects} & R & ::= c \mid x \mid R\ N \\ \text{Normal Objects} & N & ::= \lambda x.\, N \mid R \end{array}$$

We use a to range over type constants, c over object constants, and x over object variables. We follow the usual syntactic conventions: $\to$ associates to the right, and application to the left. Parentheses group subexpressions, and the scope of a λ-abstraction extends to the innermost enclosing parentheses or to the end of the expression. We allow tacit α-conversion (renaming of bound variables) and write $[M/x{:}A]N$ for the β-normal form of the result of capture-avoiding substitution of M for x in N. Constants

and variables are declared and assigned types in a signature Σ and context Γ, respectively. Neither is permitted to declare constants or variables more than once. The main judgments of the type theory are

$\Gamma \vdash_\Sigma N \Leftarrow A$ $\quad N$ has type A, and
$\Gamma \vdash_\Sigma R \Rightarrow A$ $\quad R$ has type A.

Here $N \Leftarrow A$ checks N against a given A, while $R \Rightarrow A$ synthesizes A from R or fails. In both cases we assume Σ and Γ are given. We have omitted type labels from λ-abstractions, since they are inherited from the type that a canonical object is checked against.

Returning to the representation of first-order logic, we introduce two declarations

$\mathsf{i} : \mathsf{type}$
$\mathsf{o} : \mathsf{type}$

for the types of representations of terms and formulas, respectively. For every function symbol f of arity k, we add a corresponding declaration

$$\mathsf{f} : \underbrace{\mathsf{i} \to \cdots \to \mathsf{i} \to}_{k} \mathsf{i}.$$

One of the central ideas in using a λ-calculus for representation is to represent object-language variables by meta-language variables. Through λ-abstraction at the meta-level we can properly delineate the scopes of variables bound in the object language. For simplicity, we give corresponding variables the same name in the two languages.

$$\begin{aligned} \ulcorner x \urcorner &= x \\ \ulcorner f(t_1, \ldots, t_k) \urcorner &= \mathsf{f}\, \ulcorner t_1 \urcorner \ldots \ulcorner t_k \urcorner \end{aligned}$$

Predicate symbols are dealt with like function symbols. We add a declaration

$$\mathsf{p} : \underbrace{\mathsf{i} \to \cdots \to \mathsf{i} \to}_{k} \mathsf{o}$$

for every predicate symbol p of arity k. Here are the remaining cases of the representation function.

$$\begin{aligned} \ulcorner p(t_1, \ldots, t_k) \urcorner &= \mathsf{p}\, \ulcorner t_1 \urcorner \ldots \ulcorner t_k \urcorner \\ \ulcorner A_1 \supset A_2 \urcorner &= \mathsf{imp}\, \ulcorner A_1 \urcorner\, \ulcorner A_2 \urcorner \qquad \mathsf{imp} : \mathsf{o} \to \mathsf{o} \to \mathsf{o} \\ \ulcorner \neg A \urcorner &= \mathsf{not}\, \ulcorner A \urcorner \qquad \mathsf{not} : \mathsf{o} \to \mathsf{o} \\ \ulcorner \forall x.\, A \urcorner &= \mathsf{forall}\, (\lambda x. \ulcorner A \urcorner) \qquad \mathsf{forall} : (\mathsf{i} \to \mathsf{o}) \to \mathsf{o} \end{aligned}$$

The last case in the definition introduces the concept of *higher-order abstract syntax*. If we represent variables of the object language by variables in the meta-language, then variables bound by a construct in the object language must be bound in the representation as well. The simply-typed λ-calculus has a single binding operator λ, so all variable binding is mapped to binding by λ. This idea goes back to Church's formulation of classical type theory and Martin-Löf's system of arities (Nordström et al., 1990).

This leads to the first important representation principle of logical frameworks employing higher-order abstract syntax: *Bound variable renaming in the object language is modeled by α-conversion in the meta-language.* Since we follow the variable convention in the meta-language, the variable convention in the object language is automatically supported in a framework using the representation technique above. Consequently, it cannot be used directly for binding operators for which renaming is not valid such as occur, for example, in module systems of programming languages.

The variable binding constructor "$\forall$" of the object language is translated into a second-order constructor forall in the meta-language, since delineating the scope of x introduces a function $(\lambda x.\ulcorner A\urcorner)$ of type $\mathsf{i} \to \mathsf{o}$. What does it mean to apply this function to an argument $\ulcorner t\urcorner$? This question leads to the concept of *compositionality*, a crucial property of higher-order abstract syntax. We can show by a simple induction that

$$[\ulcorner t\urcorner/x{:}\mathsf{i}]\ulcorner A\urcorner = \ulcorner [t/x]A\urcorner.$$

Note that the substitution on the left-hand side is in the framework, on the right in first-order logic. Both substitutions are defined to rename bound variables as necessary in order to avoid the capturing of variables free in t. Compositionality also plays a very important role in the representation of deductions in Section 3; we summarize it as: *Substitution in the object language is modeled by substitution in the meta-language.*

The declarations of the basic constants above are *open-ended* in the sense that we can always add further constants without destroying the validity of earlier representations. However, the definition also has an inductive character in the sense that the validity judgment of the meta-language ($\lambda^{\to}$, in this case) is defined inductively by some axioms and rules of inference. Therefore we can state and prove that there is a *compositional bijection* between well-formed formulas and normal objects of type o. Since a term or formula may have free individual variables, and they are represented by corresponding variables in the meta-language, we must take care to declare them with their proper types in the meta-language context. We refer to the particular signature with the declarations for term and formula constructors as F.

THEOREM 1 (Adequacy).

1. *We have*

$$x_1{:}\mathsf{i}, \ldots, x_n{:}\mathsf{i} \vdash_F M \Leftarrow \mathsf{i} \quad \textit{iff} \quad M = \ulcorner t \urcorner \quad \textit{for some } t,$$

   where the free variables of term t are among $x_1, \ldots, x_n$.
2. *We have*

$$x_1{:}\mathsf{i}, \ldots, x_n{:}\mathsf{i} \vdash_F M \Leftarrow \mathsf{o} \quad \textit{iff} \quad M = \ulcorner A \urcorner \quad \textit{for some } A,$$

   where the free variables of formula A are among $x_1, \ldots, x_n$.
3. *The representation function $\ulcorner \cdot \urcorner$ is a* compositional bijection *in the sense that*

$$[\ulcorner t \urcorner / x{:}\mathsf{i}] \ulcorner s \urcorner = \ulcorner [t/x]s \urcorner \quad \textit{and} \quad [\ulcorner t \urcorner / x{:}\mathsf{i}] \ulcorner A \urcorner = \ulcorner [t/x]A \urcorner$$

Proof: In one direction we proceed by an easy induction on the structure of terms and formulas. Compositionality can also be established directly by an induction on the structure of s and A, respectively.

In the other direction we carry out an induction over the structure of the derivations of $M \Leftarrow \mathsf{i}$ and $M \Leftarrow \mathsf{o}$. To prove that the representation function is a bijection, we write down its inverse on canonical forms and prove that both compositions are identity functions. 2

We summarize the main technique introduced in this section. The technique of *higher-order abstract syntax* represents object language variables by meta-language variables. It requires λ-abstraction in the meta-language in order to properly delineate the scope of bound variables, which suggests the use of the simply-typed λ-calculus as a representation language. In this approach, variable renaming is modeled by α-conversion, and capture-avoiding substitution is modeled by meta-level substitution. Representations in LF are open-ended, rather than inductive.

3. Judgments and deductions

After designing the representation of terms and formulas, the next step is to encode the axioms and inference rules of the logic under consideration. There are several styles of deductive systems which can be found in the literature, such as the axiomatic method, categorical definitions, natural deduction, or sequent calculus.

Logical frameworks are typically designed to deal particularly well with some of these systems, while being less appropriate for others. The Automath languages were designed to reflect and promote good informal

mathematical practice. It should thus be no surprise that they were particularly well-suited to systems of natural deduction. The same is true for the LF type theory, so we concentrate on the problem of representing natural deduction first. Other systems, including sequent calculi, can also be directly encoded (Pfenning, 2000).

3.1. PARAMETRIC AND HYPOTHETICAL JUDGMENTS

First, we introduce some terminology used in the presentation of deductive systems introduced with their modern meaning by Martin-Löf (1985). We will generally interpret the notions as proof-theoretic rather than semantic, since we would like to tie them closely to logical frameworks and their implementations. A *judgment* is defined by *inference rules*. An inference rule has zero or more premises and a conclusion; an axiom is an inference rule with no premises. A judgment is *evident* or *derivable* if it can be deduced using the given rules of inference. Most inference rules are *schematic* in that they contain meta-variables. We obtain *instances* of a schematic rule by replacing meta-variables with concrete expressions of the appropriate syntactic category. Each instance of an inference rule may be used in derivations. We write $\mathcal{D} :: J$ or

$$\begin{array}{c} \mathcal{D} \\ J \end{array}$$

when $\mathcal{D}$ is a derivation of judgment J. All derivations we consider must be finite.

Natural deduction further employs *hypothetical judgments*. We write

$$\begin{array}{c} \overline{}\,u \\ J_1 \\ \vdots \\ J_2 \end{array}$$

to express that judgment J_2 is derivable under hypothesis J_1 labelled u, where the vertical dots may be filled by a *hypothetical derivation*. Hypotheses have scope, that is, they may be *discharged* so that they are not available outside a given subderivation. We annotate the discharging inference with the label of the hypothesis. The meaning of a hypothetical judgment can be explained by substitution: We can substitute an arbitrary deduction $\mathcal{E} :: J_1$ for each occurrence of a hypothesis J_1 labelled u in $\mathcal{D} :: J_2$ and obtain a derivation of J_2 that no longer depends on u. We write this substitution as $[\mathcal{E}/u]\mathcal{D} :: J_2$ or two-dimensionally by writing $\mathcal{E}$ above the hypothesis justified by u. For this to be meaningful we assume that multiple occurrences

of a label annotate the same hypothesis, and that hypotheses satisfy the structural properties of *exchange* (the order in which hypotheses are made is irrelevant), *weakening* (a hypothesis need not be used) and *contraction* (a hypothesis may be used more than once).

An important related concept is that of a *parametric judgment*. Evidence for a judgment J that is parametric in a variable a is given by a derivation $\mathcal{D} :: J$ that may contain free occurrences of a. We refer to the variable a as a *parameter* and use a and b to range over parameters. We can substitute an arbitrary object O of the appropriate syntactic category for a throughout $\mathcal{D}$ to obtain a deduction $[O/a]\mathcal{D} :: [O/a]J$. Parameters also have scope and their discharge is indicated by a superscript as for hypothesis labels.

3.2. NATURAL DEDUCTION

Natural deduction is defined via a single judgment

$$\vdash^{N} A \qquad \text{formula } A \text{ is true}$$

and the mechanisms of hypothetical and parametric deductions explained in the previous section.

In natural deduction each logical symbol is characterized by its *introduction rule* or *rules* which specify how to infer a conjunction, disjunction, implication, universal quantification, etc. The *elimination rule* or *rules* for the connective then specify how we can use a conjunction, disjunction, etc. Underlying the formulation of the introduction and elimination rules is the principle of *orthogonality*: each connective should be characterized purely by its rules, and the rules should only use judgmental notions and not other logical connectives. Furthermore, the introduction and elimination rules for a logical connective cannot be chosen freely—as explained below, they should match up in order to form a coherent system. We call these conditions *local soundness* and *local completeness*.

Local soundness expresses that we should not be able to gain information by introducing a connective and immediately eliminating it. That is, if we introduce and then eliminate a connective we should be able to reach the same judgment without this detour. We show that this is possible by exhibiting a *local reduction* on derivations. The existence of a local reduction shows that the elimination rules are not too strong—they are locally sound.

Local completeness expresses that we should not lose information by introducing a connective. That is, given a judgment there is some way to eliminate its principal connective and then re-introduce it to arrive at the original judgment. We show that this is possible by exhibiting a *local*

expansion on derivations. The existence of a local expansion shows that the elimination rules are not too weak—they are locally complete.

Under the Curry-Howard isomorphism between proofs and programs (Howard, 1980), local reduction corresponds to β-reduction and local expansion corresponds to η-expansion. We express local reductions and expansions via judgments which relate derivations of the same judgment.

$$\begin{array}{lcll} \mathcal{D} & & \mathcal{D}' & \\ \vdash^N A & \Longrightarrow_R & \vdash^N A & \mathcal{D} \text{ locally reduces to } \mathcal{D}' \end{array}$$

$$\begin{array}{lcll} \mathcal{D} & & \mathcal{D}' & \\ \vdash^N A & \Longrightarrow_E & \vdash^N A & \mathcal{D} \text{ locally expands to } \mathcal{D}' \end{array}$$

In the spirit of orthogonal definitions, we proceed connective by connective, discussing introduction and elimination rules and local reductions and expansions.

Implication. To derive $\vdash^N A \supset B$ we assume $\vdash^N A$ to derive $\vdash^N B$. Written as a hypothetical judgment:

$$\dfrac{\begin{array}{c}\overline{\vdash^N A}\,u \\ \vdots \\ \vdash^N B\end{array}}{\vdash^N A \supset B}\,\supset\mathrm{I}^u$$

The hypothetical derivation describes a construction by which we can transform a derivation of $\vdash^N A$ into a derivation of $\vdash^N B$. This is accomplished by substituting the derivation of $\vdash^N A$ for every use of the hypothesis $\vdash^N A$ labelled u in the derivation of $\vdash^N B$. The elimination rule expresses just that: if we have a derivation of $\vdash^N A \supset B$ and also a derivation of $\vdash^N A$, then we can obtain a derivation of $\vdash^N B$.

$$\dfrac{\vdash^N A \supset B \qquad \vdash^N A}{\vdash^N B}\,\supset\mathrm{E}$$

The local reduction carries out the substitution of derivations explained above.

$$\dfrac{\dfrac{\begin{array}{c}\overline{\vdash^N A}\,u \\ \mathcal{D} \\ \vdash^N B\end{array}}{\vdash^N A \supset B}\,\supset\mathrm{I}^u \qquad \begin{array}{c}\mathcal{E} \\ \vdash^N A\end{array}}{\vdash^N B}\,\supset\mathrm{E} \qquad \Longrightarrow_R \qquad \begin{array}{c}\dfrac{\mathcal{E}}{\vdash^N A}\,u \\ \mathcal{D} \\ \vdash^N B\end{array}$$

The derivation on the right depends on all the hypotheses of $\mathcal{E}$ and $\mathcal{D}$ except u, for which we have substituted $\mathcal{E}$. The reduction described above may significantly increase the overall size of the derivation, since the deduction $\mathcal{E}$ is substituted for each occurrence of the assumption labeled u in $\mathcal{D}$ and may therefore be replicated.

Local expansion is specified in a similar manner.

$$\begin{array}{c}\mathcal{D}\\ \vdash^N A \supset B\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdash^N A \supset B\end{array} \qquad \dfrac{}{\vdash^N A}\,u}{\vdash^N B}\,\supset\mathrm{E}}{\vdash^N A \supset B}\,\supset\mathrm{I}^u$$

Here, u must be a new label, that is, it cannot already be used in $\mathcal{D}$.

Negation. In order to derive $\vdash^N \neg A$ we assume $\vdash^N A$ and try to derive a contradiction. This is the usual formulation, but has the disadvantage that it requires falsehood ($\bot$) as a logical symbol, thereby violating the orthogonality principle. Thus, in intuitionistic logic, one ordinarily thinks of $\neg A$ as an abbreviation for $A \supset \bot$. An alternative rule sometimes proposed assumes $\vdash^N A$ and tries to derive $\vdash^N B$ and $\vdash^N \neg B$ for some B. This also breaks the usual pattern by requiring the logical symbol we are trying to define ($\neg$) in a premise of the introduction rule. However, there is another possibility to explain the meaning of negation without recourse to implication or falsehood. We specify that $\vdash^N \neg A$ should be derivable if we can conclude $\vdash^N p$ for any formula p from the assumption $\vdash^N A$. In other words, the deduction of the premise is hypothetical in the assumption $\vdash^N A$ and parametric in the formula p.

$$\dfrac{\begin{array}{c}\dfrac{}{\vdash^N A}\,u\\ \vdots\\ \vdash^N p\end{array}}{\vdash^N \neg A}\,\neg\mathrm{I}^{p,u} \qquad\qquad \dfrac{\vdash^N \neg A \qquad \vdash^N A}{\vdash^N C}\,\neg\mathrm{E}$$

According to our intuition, the parametric judgment should be derivable if we can substitute an arbitrary concrete formula C for the parameter p and obtain a valid derivation. Thus, p may not already occur in the conclusion $\neg A$, or in any undischarged hypothesis. The reduction rule for negation follows from this interpretation and is analogous to the reduction

for implication.

$$\dfrac{\dfrac{\dfrac{}{\vdash^N A}\,u}{\vdots\,\mathcal{D}}}{\dfrac{\vdash^N p}{\vdash^N \neg A}\,\neg\mathrm{I}^{p,u} \qquad \begin{array}{c}\mathcal{E}\\ \vdash^N A\end{array}}{\vdash^N C}\,\neg\mathrm{E} \qquad \Longrightarrow_R \qquad \dfrac{\dfrac{\mathcal{E}}{\vdash^N A}\,u}{\begin{array}{c}[C/p]\mathcal{D}\\ \vdash^N C\end{array}}$$

The local expansion is also similar to that for implication.

$$\begin{array}{c}\mathcal{D}\\ \vdash^N \neg A\end{array} \qquad \Longrightarrow_E \qquad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdash^N \neg A\end{array} \qquad \dfrac{}{\vdash^N A}\,u}{\vdash^N p}\,\neg\mathrm{E}}{\vdash^N \neg A}\,\neg\mathrm{I}^{p,u}$$

Universal quantification. Under which circumstances should we be able to derive $\vdash^N \forall x.\, A$? This clearly depends on the domain of quantification. For example, if we know that x ranges over the natural numbers, then we can conclude $\vdash^N \forall x.\, A$ if we can derive $\vdash^N [0/x]A$, $\vdash^N [1/x]A$, etc. Such a rule is not effective, since it has infinitely many premises. Thus one usually uses induction principles as inference rules. However, in a general treatment of predicate logic we would like to prove statements which are true for *all* domains of quantification. Thus we can only say that $\vdash^N \forall x.\, A$ should be derivable if $\vdash^N [a/x]A$ is derivable for an arbitrary new parameter a. Conversely, if we know $\vdash^N \forall x.\, A$, we know that $\vdash^N [t/x]A$ for any term t.

$$\dfrac{\vdash^N [a/x]A}{\vdash^N \forall x.\, A}\,\forall\mathrm{I}^a \qquad\qquad \dfrac{\vdash^N \forall x.\, A}{\vdash^N [t/x]A}\,\forall\mathrm{E}$$

The superscript a is a reminder about the proviso for the introduction rule: the parameter a must be "new", that is, it may not occur in any undischarged hypothesis in the derivation of $[a/x]A$ or in $\forall x.\, A$ itself. In other words, the derivation of the premise is parametric in a. If we know that $\vdash^N [a/x]A$ is derivable for an arbitrary a, we can conclude that $\vdash^N [t/x]A$ should be derivable for any term t. Thus we have the reduction

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdash^N [a/x]A\end{array}}{\vdash^N \forall x.\, A}\,\forall\mathrm{I}^a}{\vdash^N [t/x]A}\,\forall\mathrm{E} \qquad \Longrightarrow_R \qquad \begin{array}{c}[t/a]\mathcal{D}\\ \vdash^N [t/x]A\end{array}$$

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 to be sensible, we must know that a does not already occur in A, because otherwise the conclusion of $[t/a]\mathcal{D}$ would be $[t/a][t/x]A$. The local expansion just introduces and immediately discharges the parameter.

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

Classical logic. The inference rules so far only model intuitionistic logic, and some classically true formulas such as Peirce's law $((A \supset B) \supset A) \supset A$ (for arbitrary A and B) or double negation $(\neg\neg A) \supset A$ (for arbitrary A) are not derivable. There are a number of equivalent ways to extend the system to full classical logic, typically using negation (for example, the law of excluded middle, proof by contradiction, or double negation elimination). In the fragment without disjunction or falsehood, we might choose either a rule of double negation or proof by contradiction.

$$\dfrac{\vdash^N \neg\neg A}{\vdash^N A}\,\mathsf{dbneg} \qquad\qquad \dfrac{\begin{array}{c}\overline{\vdash^N \neg A}\,u\\ \vdots\\ \vdash^N A\end{array}}{\vdash^N A}\,\mathsf{contr}^u$$

The rule for classical logic (whichever we choose to adopt) breaks the pattern of introduction and elimination rules. One can still formulate some reductions for classical derivations, but natural deduction is at heart an intuitionistic calculus. The symmetries of classical logic are better exhibited in sequent calculi.

3.3. DEDUCTIONS AS OBJECTS

In the representation of deductions we have a basic choice between simply representing derivable judgments, or giving an explicit representation of deductions as objects. There are many circumstances where we are interested in deductions as explicit objects. For example, we may want to extract functional programs from constructive (or even classical) derivations. Or we may want to implement proof transformation and presentation tools

in a theorem proving environment. If we do not trust a complex theorem prover, we may construct it so that it generates proof objects which can be independently verified. In the architecture of proof-carrying code (Necula, 1997), deductions represented in LF are attached to mobile code to certify safety (Necula, 2002). Another class of applications is the implementation of the meta-theory of the deductive systems under consideration. For example, we may want to show that natural deductions and derivations in the sequent calculus define the same theorems and exhibit translations between them. Here, we are interested in formally specifying the local reductions and expansions.

The simply-typed λ-calculus, which we used to represent the terms and formulas of first-order logic, is also a good starting point for the representation of natural deductions. As we will see below we need to refine it further in order to allow an internal validity condition for deductions. This leads us to λ^{Π}, the dependently typed λ-calculus underlying the LF logical framework (Harper et al., 1993).

We begin by introducing a new *type* nd of natural deductions. An inference rule is a constant function from deductions of the premises to a deduction of the conclusion. For example,

$$\mathsf{impe} \ : \ \mathsf{nd} \to \mathsf{nd} \to \mathsf{nd}$$

might be used to represent implication elimination. A hypothetical deduction is represented as a function from a derivation of the hypothesis to a derivation of the conclusion.

$$\mathsf{impi} \ : \ (\mathsf{nd} \to \mathsf{nd}) \to \mathsf{nd}$$

One can clearly see that this representation requires an *external* validity condition since it does not carry the information about which instance of the judgment is shown by the derivation. For example, we have

$$\vdash \mathsf{impi}\,(\lambda u.\,\mathsf{impe}\,u\,u) \Leftarrow \mathsf{nd}$$

but this term does not represent a valid natural deduction. An external validity predicate can be specified using hereditary Harrop formulas and is executable in λProlog (Felty and Miller, 1988). However, it is not *prima facie* decidable.

Fortunately, it is possible to refine the simply-typed λ-calculus so that validity of the representation of derivations becomes an *internal* property, without destroying the decidability of the type system. This is achieved by introducing *indexed types*. Consider the following encoding of the elimination rule for implication.

$$\mathsf{impe} \ : \ \mathsf{nd}\,(\mathsf{imp}\,A\,B) \to \mathsf{nd}\,A \to \mathsf{nd}\,B$$

In this specification, $\mathsf{nd}\,(\mathsf{imp}\,A\,B)$ is a *type*, the type representing derivations of $\vdash^N A \supset B$. Thus we speak of the *judgments-as-types* principle. The *type family* nd is indexed by objects of type o.

$$\mathsf{nd} \ : \ \mathsf{o} \to \mathsf{type}$$

We call $\mathsf{o} \to \mathsf{type}$ a *kind*. Secondly, we have to consider the status of the free variables A and B in the declaration. Intuitively, impe represents a whole family of constants, one for each choice of A and B. Schematic declarations like the one given above are desirable in practice, but they lead to an undecidable type checking problem (Dowek, 1993). We can recover decidability by viewing A and B as additional arguments in the representation of $\supset$E. Thus impe has four arguments representing A, B, a derivation of $A \supset B$ and a derivation of A. It returns a derivation of B. With the usual function type constructor we could only write

$$\mathsf{impe} \ : \ \mathsf{o} \to \mathsf{o} \to \mathsf{nd}\,(\mathsf{imp}\,A\,B) \to \mathsf{nd}\,A \to \mathsf{nd}\,B.$$

This does not express the dependencies between the first two arguments and the types of the remaining arguments. Thus we name the first two arguments A and B, respectively, and write

$$\mathsf{impe} \ : \ \Pi A{:}\mathsf{o}.\,\Pi B{:}\mathsf{o}.\,\mathsf{nd}\,(\mathsf{imp}\,A\,B) \to \mathsf{nd}\,A \to \mathsf{nd}\,B.$$

This is a closed type, since the *dependent function type* constructor Π binds the following variable. From the consideration above we can see that the typing rule for application of a function with dependent type should be

$$\frac{\Gamma \vdash_\Sigma R \Rightarrow \Pi x{:}A.\,B \qquad \Gamma \vdash_\Sigma N \Leftarrow A}{\Gamma \vdash_\Sigma R\,N : [N/x{:}A^-]B}\,\mathsf{app}$$

Here, A^- is the simple type that arises by erasing all dependencies and indices from A (see Section 4). For example, given a variable $p{:}\mathsf{o}$ we have

$$p{:}\mathsf{o} \vdash_\Sigma \mathsf{impe}\,(\mathsf{not}\,p)\,p \Rightarrow \mathsf{nd}\,(\mathsf{imp}\,(\mathsf{not}\,p)\,p) \to \mathsf{nd}\,(\mathsf{not}\,p) \to \mathsf{nd}\,p$$

where the signature Σ contains the declarations for formulas and inferences rules developed above. The counterexample $\mathsf{impi}\,(\lambda u.\,\mathsf{impe}\,u\,u)$ from above is now no longer well-typed: the instance of A would have to be of the form $A_1 \supset A_2$ (first occurrence of u) and simultaneously be equal to A_1 (second occurrence of u). This is clearly impossible. The rule for λ-abstraction does not change much from the simply-typed calculus.

$$\frac{\Gamma, x{:}A \vdash_\Sigma N \Leftarrow B}{\Gamma \vdash_\Sigma \lambda x.\,N \Leftarrow \Pi x{:}A.\,B}\,\mathsf{lam}$$

The variable x may now appear free in B, whereas without dependencies it could only occur free in N. Note that no type label on λ-abstractions is needed, since the given type $\Pi x{:}A.\, B$ supplies it. From these two rules it can be seen that the rules for $\Pi x{:}A.\, B$ specialize to the rules for $A \to B$ if x does not occur in B. Thus $A \to B$ is generally considered a derived notation that stands for $\Pi x{:}A.\, B$ for a variable x not free in B.

A full complement of rules for the canonical λ^{Π} type theory is given in Section 4. With dependent function types, we can now give a representation for natural deductions with an internal validity condition. This is summarized in Theorem 2 below.

Implication. The introduction rule for implication employs a hypothetical judgment. The derivation of the hypothetical judgment in the premise is represented as a function which, when applied to a derivation of A, yields a derivation of B.

$$\ulcorner\; \dfrac{\dfrac{}{\vdash^N A}\, u \quad \begin{array}{c}\mathcal{D}\\ \vdash^N B\end{array}}{\vdash^N A \supset B}\, \supset\!\mathrm{I}^u \;\urcorner \quad = \mathsf{impi}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ (\lambda u.\, \ulcorner \mathcal{D} \urcorner)$$

The assumption A labeled by u which may be used in the derivation $\mathcal{D}$ is represented by the LF variable $u{:}\mathsf{nd}\ \ulcorner A \urcorner$ which ranges over derivations of A.

$$\ulcorner\; \dfrac{}{\vdash^N A}\, u \;\urcorner = u$$

From this we can deduce the type of the impi constant.

$$\mathsf{impi}\ :\ \Pi A{:}\mathsf{o}.\, \Pi B{:}\mathsf{o}.\, (\mathsf{nd}\, A \to \mathsf{nd}\, B) \to \mathsf{nd}\, (\mathsf{imp}\, A\, B)$$

The elimination rule is simpler, since it does not involve a hypothetical judgment. The representation of a derivation ending in the elimination rule is defined by

$$\ulcorner\; \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash^N A \supset B\end{array} \qquad \begin{array}{c}\mathcal{E}\\ \vdash^N A\end{array}}{\vdash^N B}\, \supset\!\mathrm{E} \;\urcorner \quad = \mathsf{impe}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ \ulcorner \mathcal{D} \urcorner\ \ulcorner \mathcal{E} \urcorner$$

where

$\mathsf{impe}\ :\ \Pi A{:}\mathsf{o}.\,\Pi B{:}\mathsf{o}.\,\mathsf{nd}\ (\mathsf{imp}\ A\ B) \to \mathsf{nd}\ A \to \mathsf{nd}\ B.$

As an example we consider a derivation of $A \supset (B \supset A)$.

$$\dfrac{\dfrac{\dfrac{}{\vdash^N A}\,u}{\vdash^N B \supset A}\,\supset\mathrm{I}^w}{\vdash^N A \supset (B \supset A)}\,\supset\mathrm{I}^u$$

Note that the assumption $\vdash^N B$ labelled w is not used and therefore does not appear in the derivation. This derivation is represented by the LF object

$$\mathsf{impi}\ \ulcorner A \urcorner\ (\mathsf{imp}\ \ulcorner B \urcorner\ \ulcorner A \urcorner)\ (\lambda u.\,\mathsf{impi}\ \ulcorner B \urcorner\ \ulcorner A \urcorner\ (\lambda w.\,u))$$

which has type

$$\mathsf{nd}\ (\mathsf{imp}\ \ulcorner A \urcorner\ (\mathsf{imp}\ \ulcorner B \urcorner\ \ulcorner A \urcorner)).$$

This example shows clearly some redundancies in the representation of the deduction (there are many occurrences of $\ulcorner A \urcorner$ and $\ulcorner B \urcorner$). Fortunately, it is possible to analyze the types of constructors and eliminate much of this redundancy through term reconstruction (Pfenning, 1991; Necula, 2002).

Negation. The introduction and elimination rules for negation and their representation follow the pattern of the rules for implication.

$$\ulcorner\ \dfrac{\dfrac{}{\vdash^N A}\,u \quad\quad \begin{array}{c}\mathcal{D}\\ \vdash^N p\end{array}}{\vdash^N \neg A}\,\neg\mathrm{I}^{p,u}\ \urcorner \quad = \mathsf{noti}\ \ulcorner A \urcorner\ (\lambda p.\,\lambda u.\,\ulcorner \mathcal{D} \urcorner)$$

The judgment of the premise is parametric in $p{:}\mathsf{o}$ and hypothetical in $u{:}\mathsf{nd}\ulcorner A\urcorner$. It is thus represented as a function of two arguments, accepting both a formula p and a deduction of A.

$\mathsf{noti}\ :\ \Pi A{:}\mathsf{o}.\,(\Pi p{:}\mathsf{o}.\,\mathsf{nd}\ A \to \mathsf{nd}\ p) \to \mathsf{nd}\ (\mathsf{not}\ A)$

The representation of negation elimination

$$\ulcorner\ \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash^N \neg A\end{array} \quad\quad \begin{array}{c}\mathcal{E}\\ \vdash^N A\end{array}}{\vdash^N C}\,\neg\mathrm{E}\ \urcorner \quad = \mathsf{note}\ \ulcorner A \urcorner\ \ulcorner \mathcal{D} \urcorner\ \ulcorner C \urcorner\ \ulcorner \mathcal{E} \urcorner$$

leads to the following declaration

$$\mathsf{note} \;:\; \Pi A{:}\mathsf{o}.\ \mathsf{nd}\ (\mathsf{not}\ A) \to \Pi C{:}\mathsf{o}.\ \mathsf{nd}\ A \to \mathsf{nd}\ C$$

This type just inverts the second argument and result of the noti constant, which is the reason for the chosen argument order. Clearly,

$$\mathsf{note}' \;:\; \Pi A{:}\mathsf{o}.\ \Pi C{:}\mathsf{o}.\ \mathsf{nd}\ (\mathsf{not}\ A) \to \mathsf{nd}\ A \to \mathsf{nd}\ C$$

is an alternative declaration that would work just as well.

Universal quantification. Recall that $\ulcorner \forall x.\, A \urcorner = \mathsf{forall}\ (\lambda x.\ulcorner A \urcorner)$ and that the premise of the introduction rule is parametric in a.

$$\ulcorner\ \begin{array}{c} \mathcal{D} \\ \vdash^N [a/x]A \\ \hline \vdash^N \forall x.\, A \end{array} \forall \mathrm{I}^a\ \urcorner = \mathsf{foralli}\ (\lambda x.\ulcorner A \urcorner)\ (\lambda a.\ulcorner \mathcal{D} \urcorner)$$

Note that $\ulcorner A \urcorner$, the representation of A, has a free variable x which must be bound in the meta-language, so that the representing object does not have a free variable x. Similarly, the parameter a is bound at this inference and must be correspondingly bound in the meta-language. The representation determines the type of the constant foralli.

$$\mathsf{foralli} \;:\; \Pi A{:}\mathsf{i} \to \mathsf{o}.\ (\Pi a{:}\mathsf{i}.\ \mathsf{nd}\ (A\ a)) \to \mathsf{nd}\ (\mathsf{forall}\ (\lambda x.\, A\ x))$$

In an application of this constant, the argument labelled A will be $\lambda x{:}\mathsf{i}.\ulcorner A \urcorner$ and $(A\ a)$ will become $[\ulcorner a \urcorner / x{:}\mathsf{i}]\ulcorner A \urcorner$ which in turn is equal to $\ulcorner [a/x]A \urcorner$ by the compositionality of the representation.

The elimination rule does not employ a hypothetical judgment.

$$\ulcorner\ \begin{array}{c} \mathcal{D} \\ \vdash^N \forall x.\, A \\ \hline \vdash^N [t/x]A \end{array} \forall \mathrm{E}\ \urcorner = \mathsf{foralle}\ (\lambda x.\ulcorner A \urcorner)\ulcorner \mathcal{D} \urcorner\ulcorner t \urcorner$$

The substitution of t for x in A is representation by the application of the function $(\lambda x.\ulcorner A \urcorner)$ (the first argument to foralle) to $\ulcorner t \urcorner$.

$$\mathsf{foralle} \;:\; \Pi A{:}\mathsf{i} \to \mathsf{o}.\ \mathsf{nd}\ (\mathsf{forall}\ A) \to \Pi t{:}\mathsf{i}.\ \mathsf{nd}\ (A\ t)$$

We now check that

$$\ulcorner\ \begin{array}{c} \mathcal{D} \\ \vdash^N \forall x.\, A \\ \hline \vdash^N [t/x]A \end{array} \forall \mathrm{E}\ \urcorner \Leftarrow \mathsf{nd}\ \ulcorner [t/x]A \urcorner,$$

assuming that $\ulcorner\mathcal{D}\urcorner \Leftarrow \mathsf{nd}\ \ulcorner\forall x.\, A\urcorner$. This is a part in the proof of adequacy of this representation of natural deductions. At each step we verify that the arguments have the expected type and compute the type of the application.

$$\begin{array}{l}
\mathsf{foralle} \Rightarrow \Pi A{:}\mathsf{i} \to \mathsf{o}.\ \mathsf{nd}\ (\mathsf{forall}\ (\lambda x.\, A\ x)) \to \Pi t{:}\mathsf{i}.\ \mathsf{nd}\ (A\ t) \\
\mathsf{foralle}\ (\lambda x.\, \ulcorner A\urcorner) \Rightarrow \mathsf{nd}\ (\mathsf{forall}\ (\lambda x.\, \ulcorner A\urcorner)) \to \Pi t{:}\mathsf{i}.\ \mathsf{nd}\ ([t/x{:}\mathsf{i}]\ulcorner A\urcorner) \\
\mathsf{foralle}\ (\lambda x.\, \ulcorner A\urcorner)\ \ulcorner\mathcal{D}\urcorner \Rightarrow \Pi t{:}\mathsf{i}.\ \mathsf{nd}\ ([t/x{:}\mathsf{i}]\ulcorner A\urcorner) \\
\mathsf{foralle}\ (\lambda x.\, \ulcorner A\urcorner)\ \ulcorner\mathcal{D}\urcorner\ \ulcorner t\urcorner \Rightarrow \mathsf{nd}\ ([\ulcorner t\urcorner/x{:}\mathsf{i}]\ulcorner A\urcorner)) \\
\mathsf{foralle}\ (\lambda x.\, \ulcorner A\urcorner)\ \ulcorner\mathcal{D}\urcorner\ \ulcorner t\urcorner \Leftarrow \mathsf{nd}\ ([\ulcorner t\urcorner/x{:}\mathsf{i}]\ulcorner A\urcorner)
\end{array}$$

The first step follows by the nature of canonical substitution,

$$[(\lambda x.\, \ulcorner A\urcorner)/A{:}\mathsf{i} \to \mathsf{o}](A\ t) = [t/x{:}\mathsf{i}]\ulcorner A\urcorner.$$

The last step uses the rule that an atomic object of atomic type P is also canonical at type P. Furthermore, by the compositionality of the representation we have

$$[\ulcorner t\urcorner/x{:}\mathsf{i}]\ulcorner A\urcorner = \ulcorner[t/x]A\urcorner$$

which, together with the last line above, yields the desired

$$\mathsf{foralle}\ (\lambda x.\, \ulcorner A\urcorner)\ \ulcorner\mathcal{D}\urcorner\ \ulcorner t\urcorner \Leftarrow \mathsf{nd}\ (\ulcorner[t/x]A\urcorner).$$

The representation theorem relates canonical objects constructed in certain contexts to natural deductions. The restriction to canonical objects is once again crucial, as are the restrictions on the form of the context. We call the signature consisting of the declarations for first-order terms, formulas, and natural deductions ND.

THEOREM 2 (Adequacy).

1. *If $\mathcal{D}$ is a derivation of A from hypotheses $\vdash^N A_1, \ldots, \vdash^N A_n$ labelled $u_1, \ldots, u_n$, respectively, with all free individual parameters among $a_1, \ldots, a_m$ and propositional parameters among $p_1, \ldots, p_k$ then*

   $$\Gamma \vdash_{ND} \ulcorner\mathcal{D}\urcorner \Leftarrow \mathsf{nd}\ulcorner A\urcorner$$

   for $\Gamma = a_1{:}\mathsf{i}, \ldots, a_m{:}\mathsf{i}, p_1{:}\mathsf{o}, \ldots, p_k{:}\mathsf{o}, u_1{:}\mathsf{nd}\ulcorner A_1\urcorner, \ldots, u_n{:}\mathsf{nd}\ulcorner A_n\urcorner$.
2. *If $\Gamma = a_1{:}\mathsf{i}, \ldots, a_m{:}\mathsf{i}, p_1{:}\mathsf{o}, \ldots, p_k{:}\mathsf{o}, u_1{:}\mathsf{nd}\ulcorner A_1\urcorner, \ldots, u_n{:}\mathsf{nd}\ulcorner A_n\urcorner$ and*

   $$\Gamma \vdash_{ND} M \Leftarrow \mathsf{nd}\ulcorner A\urcorner$$

   then $M = \ulcorner\mathcal{D}\urcorner$ for a derivation $\mathcal{D}$ as in part 1.

3. *The representation function is a bijection, and is compositional in the sense that the following equalities hold (where* $\mathcal{E} :: \vdash^N A$*):*

$$\begin{array}{rcl} \ulcorner[t/a]\mathcal{D}\urcorner & = & [\ulcorner t\urcorner/a{:}\mathsf{i}]\ulcorner\mathcal{D}\urcorner \\ \ulcorner[C/p]\mathcal{D}\urcorner & = & [\ulcorner C\urcorner/p{:}\mathsf{o}]\ulcorner\mathcal{D}\urcorner \\ \ulcorner[\mathcal{E}/u]\mathcal{D}\urcorner & = & [\ulcorner\mathcal{E}\urcorner/u{:}\mathsf{nd}]\ulcorner\mathcal{D}\urcorner \end{array}$$

Proof: The proof proceeds by induction on the structure of natural deductions one direction and on the definition of canonical forms in the other direction. 2

Each of the rules that may be added to obtain classical logic can be easily represented with the techniques from above. They are left as an exercise to the reader.

We summarize the LF encoding of natural deductions. First, the syntax.

$$\begin{array}{lcl} \mathsf{i} & : & \mathsf{type} \\ \mathsf{o} & : & \mathsf{type} \\ \mathsf{imp} & : & \mathsf{o} \to \mathsf{o} \to \mathsf{o} \\ \mathsf{not} & : & \mathsf{o} \to \mathsf{o} \\ \mathsf{forall} & : & (\mathsf{i} \to \mathsf{o}) \to \mathsf{o} \end{array}$$

The second simplification in the concrete presentation is to omit some Π-quantifiers. Free variables in a declaration are then interpreted as a schematic variables whose quantifiers remain implicit. The types of such free variables must be determined from the context in which they appear. In practical implementations such as Twelf (Pfenning and Schürmann, 1999), type reconstruction will issue an error message if the type of free variables is ambiguous.

$$\begin{array}{lcl} \mathsf{nd} & : & \mathsf{o} \to \mathsf{type} \\ \mathsf{impi} & : & (\mathsf{nd}\, A \to \mathsf{nd}\, B) \to \mathsf{nd}\, (\mathsf{imp}\, A\, B). \\ \mathsf{impe} & : & \mathsf{nd}\, (\mathsf{imp}\, A\, B) \to \mathsf{nd}\, A \to \mathsf{nd}\, B. \\ \mathsf{noti} & : & (\Pi p{:}\mathsf{o}.\, \mathsf{nd}\, A \to \mathsf{nd}\, p) \to \mathsf{nd}\, (\mathsf{not}\, A). \\ \mathsf{note} & : & \mathsf{nd}\, (\mathsf{not}\, A) \to \Pi C{:}\mathsf{o}.\, \mathsf{nd}\, A \to \mathsf{nd}\, C. \\ \mathsf{foralli} & : & (\Pi a{:}\mathsf{i}.\, \mathsf{nd}\, (A\, a)) \to \mathsf{nd}\, (\mathsf{forall}\, (\lambda x.\, A\, x)). \\ \mathsf{foralle} & : & \mathsf{nd}\, (\mathsf{forall}\, A) \to \Pi t{:}\mathsf{i}.\, \mathsf{nd}\, (A\, t) \end{array}$$

When constants with implicitly quantified types are used, arguments corresponding to the omitted quantifiers are also left implicit. Again, in practical implementations these arguments are inferred from context. For example, the constant impi now appears to take only two arguments (of

type $\mathsf{nd}\,A$ and $\mathsf{nd}\,B$ for some A and B) rather than four, like the fully explicit declaration

$$\mathsf{impi} \;:\; \Pi A{:}\mathsf{o}.\,\Pi B{:}\mathsf{o}.\,(\mathsf{nd}\,A \to \mathsf{nd}\,B) \to \mathsf{nd}\,(\mathsf{imp}\,A\,B).$$

The derivation of $A \supset (B \supset A)$ from above has this very concise representation:

$$\mathsf{impi}\,(\lambda u.\,\mathsf{impi}\,(\lambda w.\,u)) \Leftarrow \mathsf{nd}(\mathsf{imp}\,A\,(\mathsf{imp}\,B\,A))$$

To recover classical logic, we can add either of the following declarations to the signature, modeling the two rules previously introduced.

$$\begin{array}{lcl} \mathsf{dbneg} & : & \mathsf{nd}\,(\mathsf{not}\,(\mathsf{not}\,A)) \to \mathsf{nd}\,A. \\ \mathsf{contr} & : & (\mathsf{nd}\,(\mathsf{not}\,A) \to \mathsf{nd}\,A) \to \mathsf{nd}\;A. \end{array}$$

In summary, the basic representation principle underlying LF is the representation of judgments as types. A deduction of a judgment J is represented as a canonical object N whose type is the representation of J. This basic scheme is extended to represent hypothetical judgments as simple function types and parametric judgments as dependent function types. This encoding reduces the question of validity for a derivation to the question of well-typedness for its representation. Since type-checking in the LF type theory is decidable, the validity of derivations has been internalized as a decidable property in the logical framework.

3.4. HIGHER-LEVEL JUDGMENTS

Next we turn to the local reduction judgment for natural deductions introduced in Section 3.2.

$$\begin{array}{c} \mathcal{D} \\ \vdash^{N} A \end{array} \quad \Longrightarrow_R \quad \begin{array}{c} \mathcal{D}' \\ \vdash^{N} A \end{array}$$

Recall that this judgment witnesses the local soundness of the elimination rules with respect to the introduction rules. We refer to this as a *higher-level judgment* since it relates derivations. The representation techniques underlying LF support this directly, since deductions are represented as objects which can in turn index type families representing higher-level judgments.

In this particular example, reduction is defined only by axioms, one each for implication, negation, and universal quantification. The representing type family in LF must be indexed by the representation of two deductions $\mathcal{D}$ and $\mathcal{D}'$, and consequently also by the representation of A. This shows

that there may be dependencies between indices to a type family so that we need a dependent constructor Π for kinds in order to represent judgments relating derivations.

$$\mathsf{redl} \ : \ \Pi A{:}\mathsf{o}.\ \mathsf{nd}\ A \to \mathsf{nd}\ A \to \mathsf{type}.$$

As in the representation of inference rules in Section 3.3, we omit the explicit quantifier on A and determine A from context.

$$\mathsf{redl} \ : \ \mathsf{nd}\ A \to \mathsf{nd}\ A \to \mathsf{type}.$$

We show the representation of the reduction rules for each connective in turn.

Implication. This reduction involves a substitution of a derivation for an assumption.

$$\dfrac{\dfrac{\dfrac{\overline{\vdash^N A}\,u}{\begin{array}{c}\mathcal{D}\\ \vdash^N B\end{array}}}{\vdash^N A \supset B}\supset\mathrm{I}^u \qquad \begin{array}{c}\mathcal{E}\\ \vdash^N A\end{array}}{\vdash^N B}\supset\mathrm{E} \qquad \Longrightarrow_R \qquad \dfrac{\begin{array}{c}\mathcal{E}\\ \vdash^N A\end{array}}{\begin{array}{c}\mathcal{D}\\ \vdash^N B\end{array}}\,u$$

The representation of the left-hand side is

$$\mathsf{impe}\ (\mathsf{impi}\ (\lambda u.\ D\ u))\ E$$

where $E = \ulcorner\mathcal{E}\urcorner \Leftarrow \mathsf{nd}\,A$ and $D = (\lambda u.\ulcorner\mathcal{D}\urcorner) \Leftarrow \mathsf{nd}\,A \to \mathsf{nd}\,B$. The derivation on the right-hand side can be written more succinctly as $[\mathcal{E}/u]\mathcal{D}$. Compositionality of the representation (Theorem 2, part 3) yields

$$\ulcorner[\mathcal{E}/u]\mathcal{D}\urcorner = [\ulcorner\mathcal{E}\urcorner/u{:}\mathsf{nd}]\ulcorner\mathcal{D}\urcorner.$$

Thus we can formulate the rule concisely as

$$\mathsf{redl\_imp} \ : \ \mathsf{redl}\ (\mathsf{impe}\ (\mathsf{impi}\ (\lambda u.\ D\ u))\ E)\ (D\ E)$$

Negation. This is similar to implication. The required substitution of C for p in $\mathcal{D}$ is implemented by application and β-reduction at the meta-level.

$$\dfrac{\dfrac{\dfrac{\overline{\vdash^N A}\,u}{\begin{array}{c}\mathcal{D}\\ \vdash^N p\end{array}}}{\vdash^N \neg A}\neg\mathrm{I}^{p,u} \qquad \begin{array}{c}\mathcal{E}\\ \vdash^N A\end{array}}{\vdash^N C}\neg\mathrm{E} \qquad \Longrightarrow_R \qquad \dfrac{\begin{array}{c}\mathcal{E}\\ \vdash^N A\end{array}}{\begin{array}{c}[C/p]\mathcal{D}\\ \vdash^N C\end{array}}\,u$$

$\mathsf{redl\_not} \ : \ \mathsf{redl}\ (\mathsf{note}\ (\mathsf{noti}\ (\lambda p.\, \lambda u.\, D\ p\ u))\ C\ E)\ (D\ C\ E).$

Universal quantification. The universal introduction rule involves a parametric judgment. Consequently, the substitution to be carried out during reduction replaces a parameter by a term.

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

In the representation we once again exploit the compositionality.

$$\ulcorner [t/a]\mathcal{D} \urcorner = [\ulcorner t \urcorner / a{:}\mathsf{i}] \ulcorner \mathcal{D} \urcorner.$$

This gives rise to the declaration

$\mathsf{redl\_forall} \ : \ \mathsf{redl}\ (\mathsf{foralle}\ (\mathsf{foralli}\ (\lambda a.\, D\ a))\ T)\ (D\ T).$

The adequacy theorem for this encoding states that canonical LF objects of type $\mathsf{redl}\ \ulcorner \mathcal{D} \urcorner\ \ulcorner \mathcal{D}' \urcorner$ constructed over the appropriate signature and in an appropriate parameter context are in bijective correspondence with derivations of $\mathcal{D} \Longrightarrow_R \mathcal{D}'$. We leave the precise formulation and simple proof to the diligent reader.

The encoding of the local expansions employs the same techniques. We summarize it below without going into further detail.

$$\begin{array}{ll}
\mathsf{expl} & : \ \Pi A{:}o.\, \mathsf{nd}\ A \to \mathsf{nd}\ A \to \mathsf{type}. \\
\mathsf{expl_imp} & : \ \mathsf{expl}\ (\mathsf{imp}\ A\ B)\ D\ (\mathsf{impi}\ (\lambda u.\, \mathsf{impe}\ D\ u)). \\
\mathsf{expl_not} & : \ \mathsf{expl}\ (\mathsf{not}\ A)\ D\ (\mathsf{noti}\ (\lambda p.\, \lambda u.\, \mathsf{note}\ D\ p\ u)). \\
\mathsf{expl_forall} & : \ \mathsf{expl}\ (\mathsf{forall}\ (\lambda x.\, A\ x))\ D\ (\mathsf{foralli}\ (\lambda a.\, \mathsf{foralle}\ D\ a)).
\end{array}$$

In summary, the representation of higher-level judgments continues to follow the *judgments-as-types* technique. The expressions related by higher-level judgments are now deductions and therefore dependently typed in the representation. Substitution at the level of deductions is implemented by substitution at the meta-level, taking advantage of the compositionality of the representation.

4. A dependently typed λ-calculus

In this section we summarize a recent formulation of the dependently λ-calculus λ^{Π} allowing only canonical forms (Watkins et al., 2002). This

avoids an explicit notion of definitional equality (Harper et al., 1993), which is not required for applications of λ^Π as a logical framework. Related systems have been advocated by de Bruijn (1993) and Felty (1991). See Watkins et al. (2002) for further details and properties of this formulation.

λ^Π is predicative calculus with three levels: kinds, families, and objects. We also define signatures and contexts as they are needed for the judgments.

$$
\begin{array}{lll}
\text{Normal Kinds} & K & ::= \ \mathsf{type} \mid \Pi x{:}A.\, K \\
\text{Atomic Types} & P & ::= \ a \mid P\ N \\
\text{Normal Types} & A & ::= \ P \mid \Pi x{:}A_1.\, A_2 \\
\text{Atomic Objects} & R & ::= \ c \mid x \mid R\ N \\
\text{Normal Objects} & N & ::= \ \lambda x.\, N \mid R \\
\text{Signatures} & \Sigma & ::= \ \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\
\text{Contexts} & \Gamma & ::= \ \cdot \mid \Gamma, x{:}A
\end{array}
$$

We write a for type family constants and c for object constants, both declared in signatures Σ with their kind and type, respectively. Variables x are declared in contexts with their type. We make the uniform assumption that no constant or variable may be declared more than once in a signature or context, respectively. We also allow tacit renaming of variables bound by $\Pi x{:}A \ldots$ and $\lambda x \ldots$. As usual, we avoid an explicit non-dependent function type by thinking of $A \to B$ as an abbreviation for $\Pi x{:}A.\, B$ where x does not occur in B, and similarly for $A \to K$.

From the point of view of natural deduction, atomic objects are composed of destructors corresponding to elimination rules, while normal objects are built from constructors corresponding to introduction rules. The typing rules are *bi-directional* which mirrors the syntactic structure of normal forms: we check a normal object against a type, and we synthesize a type for an atomic object. We write $U \Leftarrow V$ to indicate that U is checked against a given V (which we assume is valid), and $U \Rightarrow V$ to indicate that U synthesizes a V (which we prove is valid).

$$
\begin{array}{ll}
\Gamma \vdash_\Sigma K \Leftarrow \mathsf{kind} & K \text{ is a valid kind} \\
\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} & A \text{ is a valid type} \\
\Gamma \vdash_\Sigma P \Rightarrow K & P \text{ is atomic of kind } K \\
\Gamma \vdash_\Sigma N \Leftarrow A & N \text{ is normal of type } A \\
\Gamma \vdash_\Sigma R \Rightarrow A & R \text{ is atomic of type } A \\
\vdash \Sigma\ \mathsf{Sig} & \Sigma \text{ is a valid signature} \\
\vdash_\Sigma \Gamma\ \mathsf{Ctx} & \Gamma \text{ is a valid context}
\end{array}
$$

In one rule, we write $A \equiv A'$ for syntactic equality of normal types modulo α-conversion. This is to emphasize the flow of information during type-checking.

$$\frac{}{\vdash \cdot \ \mathsf{Sig}} \qquad \frac{\vdash \Sigma \ \mathsf{Sig} \qquad \cdot \vdash_\Sigma K \Leftarrow \mathsf{kind}}{\vdash \Sigma, a{:}K \ \mathsf{Sig}} \qquad \frac{\vdash \Sigma \ \mathsf{Sig} \qquad \cdot \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash \Sigma, c{:}A \ \mathsf{Sig}}$$

$$\frac{}{\vdash \cdot \ \mathsf{Ctx}} \qquad \frac{\vdash_\Sigma \Gamma \ \mathsf{Ctx} \qquad \Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash_\Sigma \Gamma, x{:}A \ \mathsf{Ctx}}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{type} \Leftarrow \mathsf{kind}} \qquad \frac{\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma K \Leftarrow \mathsf{kind}}{\Gamma \vdash_\Sigma \Pi x{:}A.\, K \Leftarrow \mathsf{kind}}$$

$$\frac{}{\Gamma \vdash_\Sigma a \Rightarrow \Sigma(a)} \qquad \frac{\Gamma \vdash_\Sigma P \Rightarrow \Pi x{:}A.\, K \qquad \Gamma \vdash_\Sigma N \Leftarrow A}{\Gamma \vdash_\Sigma P\ N \Rightarrow [N/u{:}A^-]K}$$

$$\frac{\Gamma \vdash_\Sigma P \Rightarrow \mathsf{type}}{\Gamma \vdash_\Sigma P \Leftarrow \mathsf{type}} \qquad \frac{\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma B \Leftarrow \mathsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.\, B \Leftarrow \mathsf{type}}$$

$$\frac{}{\Gamma \vdash_\Sigma c \Rightarrow \Sigma(c)} \qquad \frac{}{\Gamma \vdash_\Sigma x \Rightarrow \Gamma(x)} \qquad \frac{\Gamma \vdash_\Sigma R \Rightarrow \Pi x{:}A.\, B \qquad \Gamma \vdash_\Sigma N \Leftarrow A}{\Gamma \vdash_\Sigma R\ N \Rightarrow [N/x{:}A^-]B}$$

$$\frac{\Gamma \vdash_\Sigma N \Rightarrow A' \qquad A' \equiv A}{\Gamma \vdash_\Sigma N \Leftarrow A} \qquad \frac{\Gamma, x{:}A \vdash_\Sigma N \Leftarrow B}{\Gamma \vdash_\Sigma \lambda x.\, N \Leftarrow \Pi x{:}A.\, B}$$

In order to define canonical substitution inductively, we erase all dependences and indices from a type to obtain a simple type τ.

$$\begin{array}{lcl} (a)^- & = & a \\ (P\ N)^- & = & P^- \\ (\Pi x{:}A.\, B)^- & = & A^- \to B^- \end{array}$$

The canonical substitution $[N/x{:}\tau]B$ and $[N/x{:}\tau]K$ returns a normal type or kind, respectively. It is defined inductively, first on the structure of the simply-typed erasure A^- of A and then the structure of B and K, respectively. Modulo the proof of termination, we can also think of it as the

β-normal form of $[N/x]B$ and $[N/x]K$, respectively. For details of the two approaches, the reader may consult Watkins et al. (2002) and Felty (1991).

$$
\begin{array}{lcl}
[N_0/x{:}\tau](\mathsf{type}) & = & \mathsf{type} \\
[N_0/x{:}\tau](\Pi y{:}A.\, K) & = & \Pi y{:}[N_0/x{:}\tau]A.\, [N_0/x{:}\tau]K \\
[N_0/x{:}\tau](a) & = & a \\
[N_0/x{:}\tau](P\ N) & = & ([N_0/x{:}\tau]P)\ ([N_0/x{:}\tau]N) \\
[N_0/x{:}\tau](\Pi y{:}A.\, B) & = & \Pi y{:}[N_0/x{:}\tau]A.\, [N_0/x{:}\tau]B \\
[N_0/x{:}\tau](\lambda y.\, N) & = & \lambda y.\, [N_0/x{:}\tau]N \\
[N_0/x{:}\tau](R) & = & [N_0/x{:}\tau]^r(R) \quad \text{or} \quad [N_0/x{:}\tau]^\beta(R) \\
[N_0/x{:}\tau]^r(c) & = & c \\
[N_0/x{:}\tau]^r(x) & & \text{undefined} \\
[N_0/x{:}\tau]^r(y) & = & y \qquad \text{provided } x \neq y \\
[N_0/x{:}\tau]^r(R\ N) & = & ([N_0/x{:}\tau]^r R)\ ([N_0/x{:}\tau]N) \\
[N_0/x{:}\tau]^\beta(c) & & \text{undefined} \\
[N_0/x{:}\tau]^\beta(x) & = & (N_0 : \tau) \\
[N_0/x{:}\tau]^\beta(y) & & \text{undefined provided } x \neq y \\
[N_0/x{:}\tau]^\beta(R\ N) & = & ([[N_0/x{:}\tau]N/y{:}\tau_1]N' : \tau_2) \\
 & & \quad \text{where } [N_0/x{:}\tau]^\beta(R) = (\lambda y.\, N' : \tau_1 \to \tau_2)
\end{array}
$$

Note that for all atomic terms R, either a case for $[N/x{:}\tau]^r(R)$ or $[N/x{:}\tau]^\beta(R)$ applies, depending on whether the head of R is x or not. Furthermore, if $[N/x{:}\tau]^\beta(R) = (N : \tau')$ then τ' is a subexpression of τ. Hence canonical substitution is a terminating function. Decidability of the LF type theory is then a straightforward consequence. Furthermore, cut is admissible in the sense that if $\Gamma \vdash_\Sigma N_0 \Leftarrow A$ and $\Gamma, x{:}A \vdash_\Sigma N \Leftarrow C$ then $\Gamma \vdash_\Sigma [N_0/x{:}A^-]N \Leftarrow [N_0/x{:}A^-]C$ (Watkins et al., 2002).

5. Conclusion

We have provided an introduction to the techniques of logical frameworks with an emphasis on LF which is based on the dependently typed λ-calculus λ^Π. We now summarize the basic choices that arise in the design of logical frameworks.

Strong vs. weak frameworks. De Bruijn, the founder of the field of logical frameworks, argues (de Bruijn, 1991) that logical frameworks should be foundationally uncommitted and as weak as possible. This allows simple proofs of adequacy for encodings, efficient checking of the correctness of derivations, and allows effective algorithms for unification and proof search

in the framework which are otherwise difficult to design (for example, in the presence of iterated inductive definitions). This is also important if we use explicit proofs as a means to increase confidence in the results of a theorem prover: the simpler the logical framework, the more trusted its implementation is likely to be.

Inductive representations vs. higher-order abstract syntax. This is related to the previous question. Inductive representations of logics are supported in various frameworks and type theories not explicitly designed as logical frameworks. They allow a formal development of the meta-theory of the deductive system in question, but the encodings are less direct than for frameworks employing higher-order abstract syntax and functional representations of hypothetical derivations. Present work on combining advantages of both either employ reflection or formal meta-reasoning about the logical framework itself (McDowell and Miller, 1997; Schürmann, 2000).

Logical vs. type-theoretic meta-languages. A logical meta-language such as one based on hereditary Harrop formulas encodes judgments as propositions. Search for a derivation in an object logic is reduced to proof search in the meta-logic. In addition, type-theoretical meta-languages such as LF offer a representation for derivations as objects. Checking the correctness of a derivation is reduced to type-checking in the meta-language. This is a decidable property that enables the use of a logical framework for applications such as proof-carrying code, where an explicit representation for deductions is required (Necula, 2002).

Framework extensions. Logical framework languages can be assessed along many dimensions, as the discussions above indicate. Three of the most important concerns are how directly object languages may be encoded, how easy it is to prove the adequacies of these encodings, and how simple the proof checker for a logical framework can be. A great deal of practical experience has been accumulated, for example, through the use of λProlog, Isabelle, and Elf. These experiments have also identified certain shortcomings in the logical frameworks. Perhaps the most important one is the treatment of substructural logics, or languages with an inherent notion of store or concurrency. Representation of such object languages is possible, but not as direct as one might wish. The logical frameworks Forum (Miller, 1994), linear LF (Cervesato and Pfenning, 1997) and CLF (Watkins et al., 2002) have been designed to overcome these shortcomings by providing linearity intrinsically. Other extensions by subtyping, module constructs, constraints, etc. have also been designed, but their discussion is beyond the scope of this introduction.

Further reading. There have been numerous case studies and applications carried out with the aid of logical frameworks or generic theorem provers, too many to survey them here. The principal application areas lie in the theory of programming languages and logics, reasoning about specifications, programs, and protocols, and the formalization of mathematics. We refer the interested reader to (Pfenning, 1996) for some further information on applications of logical frameworks. The handbook article (Pfenning, 2001b) provides more detailed development of LF and includes some material on meta-logical frameworks. A survey with deeper coverage of modal logics and inductive definitions can be found in (Basin and Matthews, 2001). The textbook (Pfenning, 2001a) provides a gentler and more thorough introduction to the pragmatics of the LF logical framework and its use for the study of programming languages.

References

Altenkirch, T., V. Gaspes, B. Nordström, and B. von Sydow: 1994, 'A User's Guide to ALF'. Chalmers University of Technology, Sweden.

Barendregt, H. P.: 1980, *The Lambda-Calculus: Its Syntax and Semantics*. North-Holland.

Basin, D. and S. Matthews: 1996, 'Structuring Metatheory on Inductive Definitions'. In: M. McRobbie and J. Slaney (eds.): *Proceedings of the 13th International Conference on Automated Deduction (CADE-13)*. New Brunswick, New Jersey, pp. 171–185, Springer-Verlag LNAI 1104.

Basin, D. and S. Matthews: 2001, 'Logical Frameworks'. In: D. Gabbay and F. Guenthner (eds.): *Handbook of Philosophical Logic*. Kluwer Academic Publishers, 2nd edition. In preparation.

Basin, D. A. and R. L. Constable: 1993, 'Metalogical Frameworks'. In: G. Huet and G. Plotkin (eds.): *Logical Environments*. Cambridge University Press, pp. 1–29.

Cervesato, I. and F. Pfenning: 1996, 'A Linear Logical Framework'. In: E. Clarke (ed.): *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*. New Brunswick, New Jersey, pp. 264–275, IEEE Computer Society Press.

Cervesato, I. and F. Pfenning: 1997, 'Linear Higher-Order Pre-Unification'. In: G. Winskel (ed.): *Proceedings of the Twelfth Annual Sumposium on Logic in Computer Science (LICS'97)*. Warsaw, Poland, pp. 422–433, IEEE Computer Society Press.

de Bruijn, N.: 1968, 'The Mathematical Language AUTOMATH, Its Usage, and Some of Its Extensions'. In: M. Laudet (ed.): *Proceedings of the Symposium on Automatic Demonstration*. Versailles, France, pp. 29–61, Springer-Verlag LNM 125.

de Bruijn, N.: 1991, 'A Plea for Weaker Frameworks'. In: G. Huet and G. Plotkin (eds.): *Logical Frameworks*. pp. 40–67, Cambridge University Press.

de Bruijn, N.: 1993, 'Algorithmic Definition of Lambda-Typed Lambda Calculus'. In: G. Huet and G. Plotkin (eds.): *Logical Environment*. Cambridge University Press, pp. 131–145.

Dowek, G.: 1993, 'The Undecidability of Typability in the Lambda-Pi-Calculus'. In: M. Bezem and J. Groote (eds.): *Proceedings of the International Conference on Typed Lambda Calculi and Applications*. Utrecht, The Netherlands, pp. 139–145, Springer-Verlag LNCS 664.

Eriksson, L.-H.: 1994, 'Pi: An Interactive Derivation Editor for the Calculus of Partial Inductive Definitions'. In: A. Bundy (ed.): *Proceedings of the 12th International Conference on Automated Deduction*. Nancy, France, pp. 821–825, Springer Verlag LNAI 814.

Feferman, S.: 1988, 'Finitary Inductive Systems'. In: R. Ferro (ed.): *Proceedings of Logic Colloquium '88*. Padova, Italy, pp. 191–220, North-Holland.

Felty, A.: 1991, 'Encoding Dependent Types in an Intuitionistic Logic'. In: G. Huet and G. D. Plotkin (eds.): *Logical Frameworks*. pp. 214–251, Cambridge University Press.

Felty, A. and D. Miller: 1988, 'Specifying Theorem Provers in a Higher-Order Logic Programming Language'. In: E. Lusk and R. Overbeek (eds.): *Proceedings of the Ninth International Conference on Automated Deduction*. Argonne, Illinois, pp. 61–80, Springer-Verlag LNCS 310.

Gabbay, D. M.: 1994, 'Classical vs Non-Classical Logic'. In: D. Gabbay, C. Hogger, and J. Robinson (eds.): *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 2. Oxford University Press, Chapt. 2.6.

Harper, R., F. Honsell, and G. Plotkin: 1993, 'A Framework for Defining Logics'. *Journal of the Association for Computing Machinery* **40**(1), 143–184.

Howard, W. A.: 1980, 'The formulae-as-types notion of construction'. In: J. P. Seldin and J. R. Hindley (eds.): *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, pp. 479–490. Hitherto unpublished note of 1969.

Martì-Oliet, N. and J. Meseguer: 1993, 'Rewriting Logic as a Logical and Semantical Framework'. Technical Report SRI-CSL-93-05, SRI International.

Martin-Löf, P.: 1980, 'Constructive Mathematics and Computer Programming'. In: *Logic, Methodology and Philosophy of Science VI*. pp. 153–175, North-Holland.

Martin-Löf, P.: 1985, 'On the Meanings of the Logical Constants and the Justifications of the Logical Laws'. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena. Reprinted in the *Nordic Journal of Philosophical Logic*, **1**(1), 11-60, 1996.

McDowell, R. and D. Miller: 1997, 'A Logic for Reasoning with Higher-Order Abstract Syntax'. In: G. Winskel (ed.): *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*. Warsaw, Poland, pp. 434–445, IEEE Computer Society Press.

Miller, D.: 1994, 'A Multiple-Conclusion Meta-Logic'. In: S. Abramsky (ed.): *Ninth Annual Symposium on Logic in Computer Science*. Paris, France, pp. 272–281, IEEE Computer Society Press.

Nadathur, G. and D. Miller: 1988, 'An Overview of λProlog'. In: K. A. Bowen and R. A. Kowalski (eds.): *Fifth International Logic Programming Conference*. Seattle, Washington, pp. 810–827, MIT Press.

Necula, G. C.: 1997, 'Proof-Carrying Code'. In: N. D. Jones (ed.): *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*. Paris, France, pp. 106–119, ACM Press.

Necula, G. C.: 2002, 'Proof-Carrying Code: Design and Implementation'. *This volume*. Kluwer Academic Publishers.

Nordström, B., K. Petersson, and J. M. Smith: 1990, *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press.

Paulson, L. C.: 1986, 'Natural Deduction as Higher-order Resolution'. *Journal of Logic Programming* **3**, 237–258.

Pfenning, F.: 1991, 'Logic Programming in the LF Logical Framework'. In: G. Huet and G. Plotkin (eds.): *Logical Frameworks*. pp. 149–181, Cambridge University Press.

Pfenning, F.: 1996, 'The Practice of Logical Frameworks'. In: H. Kirchner (ed.): *Proceed-*

ings of the Colloquium on Trees in Algebra and Programming. Linköping, Sweden, pp. 119–134, Springer-Verlag LNCS 1059. Invited talk.

Pfenning, F.: 2000, 'Structural Cut Elimination I. Intuitionistic and Classical Logic'. *Information and Computation* **157**(1/2), 84–141.

Pfenning, F.: 2001a, *Computation and Deduction*. Cambridge University Press. In preparation. Draft from April 1997 available electronically.

Pfenning, F.: 2001b, 'Logical Frameworks'. In: A. Robinson and A. Voronkov (eds.): *Handbook of Automated Reasoning*. Elsevier Science and MIT Press, Chapt. 16, pp. 977–1061. In press.

Pfenning, F. and C. Schürmann: 1999, 'System Description: Twelf — A Meta-Logical Framework for Deductive Systems'. In: H. Ganzinger (ed.): *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*. Trento, Italy, pp. 202–206, Springer-Verlag LNAI 1632.

Schürmann, C.: 2000, 'Automating the Meta Theory of Deductive Systems'. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University. Available as Technical Report CMU-CS-00-146.

Watkins, K., I. Cervesato, F. Pfenning, and D. Walker: 2002, 'A Concurrent Logical Framework I: Judgments and Properties'. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University. Forthcoming.