CHAPTER 17

Logical Frameworks

Frank Pfenning

SECOND READERS: Robert Harper, Don Sannella, and Jan Smith.

Contents

HANDBOOK OF AUTOMATED REASONING
Edited by Alan Robinson and Andrei Voronkov

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 addition, general reasoning systems have been used to study deductions as mathematical objects, without specific support for the domain of deductive systems.

In this chapter we highlight the major themes, concepts, and design choices for logical frameworks and provide pointers to the literature for further reading. We concentrate on systems designed specifically as frameworks and among them on those most immediately based on deduction: hereditary Harrop formulas (implemented in λProlog and Isabelle) and the LF type theory (implemented in Elf). We briefly mention other approaches below and discuss them in more detail in Section 8.

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 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, de Bruijn 1980, Nederpelt, Geuvers and de Vrijer 1994] 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* [Jutting 1977]. 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, Martin-Löf 1985*a*, Martin-Löf 1985*b*]. On the one hand, this led to a number of systems for constructive mathematics and the extraction of functional programs from constructive proofs (beginning with Petersson's implementation [Petersson 1982], followed by Nuprl [*Nuprl* 1999, Constable et al. 1986], Coq [*Coq* 1999, Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring and Werner 1993], PX [Hayashi and Nakano 1988], and LEGO [*LEGO* 1998, Luo and Pollack 1992, Pollack 1994]). On the other hand, it strongly influenced the design of LF [Harper, Honsell and Plotkin 1987, Harper, Honsell and Plotkin 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, Paulson 1989, Nipkow and Paulson 1992] and logic programming languages [Nadathur and Miller 1988, Miller, Nadathur, Pfenning and Scedrov 1991]. The type-theoretic and logic programming approaches were later combined in the Elf language [Pfenning 1989, Pfenning 1991*a*]. 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 Section 8.3). The mid-nineties saw renewed activity with implementations of frameworks based on inductive definitions such as FS_0 [Feferman 1988, Matthews, Smaill and Basin 1993, Basin and Matthews 1996] and ALF [Nordström 1993, Altenkirch, Gaspes, Nordström and von Sydow 1994], partial inductive definitions [Hallnäs 1991, Eriksson 1993*a*, Eriksson 1994], substructural frameworks [Schroeder-Heister 1991, Girard 1993, Miller 1994, Cervesato and Pfenning 1996, Cervesato 1996], rewriting logic [Martì-Oliet and Meseguer 1993, Borovanský, Kirchner, Kirchner, Moreau and Ringeissen 1998], and labelled deductive systems [Gabbay 1994, Basin, Matthews and Viganò 1998, Gabbay 1996]. A full discussion of these is beyond the scope of this chapter—the reader can find some brief remarks in Section 8.

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. We therefore consider them here and discuss issues related to meta-theoretic reasoning in Section 5. Systems not based on type theory are sometimes called *general logics*. We do not attempt to delineate precisely what characterizes general logics as a special case of logical frameworks, but we point out some methodological differences between approaches rooted in type theory and logic throughout this chapter. They are summarized in Section 8.

The remainder of this chapter follows the tasks which arise in a typical application of a logical framework: *specification*, *search*, and *meta-theory*. As an example we pick a fragment of predicate logic. In Section 2 we introduce techniques for the representation of formulas and other expressions of a given object logic. Section 3 treats the representation of judgments and legal deductions. These two sections therefore illustrate how logical frameworks support specification of deductive systems. Section 4 sketches generic principles underlying proof search and how they are realized in logical frameworks. It therefore covers reasoning *within* deductive systems. Section 5 discusses approaches for formal reasoning *about* the properties of logical systems. Sections 6 and 7 summarize the formal definitions underlying the frameworks under consideration in this chapter. We conclude with remarks about current lines of research and applications in Section 8.

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 this chapter we consider formulations of intuitionistic and classical first-order logic. In order to keep this chapter to a manageable length, 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, [Felty and Miller 1988, Paulson 1990, Harper et al. 1993, Pfenning 2001]).

Our fragment of first-order logic is constructed from individual variables, function symbols, and predicate symbols 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.

Function symbols	f^k		
Predicate symbols	p^k		
Variables	x		
Terms	t	$::=$	$x \mid f^k(t_1,\ldots,t_k)$
Atoms	P	$::=$	$p^k(t_1,\ldots,t_k)$
Formulas	A	$::=$	$P \mid A_1 \supset A_2 \mid \neg A \mid \forall x.\, A$

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 goes back to Church and Rosser [Church and Rosser 1936]) 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. Uni-typed representations

As the archetypical untyped representation language we choose first-order terms themselves. Actually, it is more appropriate to think of it as a *uni-typed* language, that is, a language with a single type of individuals. For each function symbol f we have a corresponding function symbol c_f of the same arity in the representation. Similarly, each predicate symbol p is represented by a constant c_p. The representation of variables is more complex, since there are infinitely many of them. For simplicity, we assume variables are enumerated and the nth variable x_n is represented by $\mathsf{var}(n)$, where the natural numbers n are either meta-language constants or constructed from constants for zero and successor. We write $\ulcorner - \urcorner$ for the representation function which maps expressions of an object language to objects in the meta-language. We use $\mathsf{sans\text{-}serif\ font}$ for constants in various logical frameworks we consider.

$$\begin{array}{rcl}
\ulcorner x_n \urcorner &=& \mathsf{var}(n)\\
\ulcorner f^k(t_1,\ldots,t_k)\urcorner &=& \mathsf{c}^k_f(\ulcorner t_1\urcorner,\ldots,\ulcorner t_k\urcorner)\\
\ulcorner p^k(t_1,\ldots,t_k)\urcorner &=& \mathsf{c}^k_p(\ulcorner t_1\urcorner,\ldots,\ulcorner t_k\urcorner)\\
\ulcorner A \supset B\urcorner &=& \mathsf{imp}(\ulcorner A\urcorner,\ulcorner B\urcorner)\\
\ulcorner \neg A\urcorner &=& \mathsf{not}(\ulcorner A\urcorner)\\
\ulcorner \forall x.\, A\urcorner &=& \mathsf{forall}(\ulcorner x\urcorner,\ulcorner A\urcorner)
\end{array}$$

However, our task is not yet complete: we need to be able to check, for example, if a given meta-language term represents a formula. For this we use Horn clauses to axiomatize the atomic proposition $\mathsf{formula}(t)$ which expresses that the meta-language term t represents a formula of the object language. This requires several auxiliary predicates to recognize representations of variables and terms. The specification below is effective in the sense that it can be executed in pure Prolog to check if a given term represents a well-formed formula. For our purposes, we think of Horn clauses as generated by the following grammar.

$$\text{Horn clauses} \quad D \quad ::= \quad P \mid \top \mid D_1 \wedge D_2 \mid P_1 \wedge \ldots \wedge P_n \supset P \mid \forall x.\, D$$

where P stands for atomic propositions and $\top$ stands for the true proposition. We refer to a collection of closed Horn clauses as a *Horn theory* and write $T \vDash^{H} P$ if the Horn theory T entails P. Natural numbers are represented in unary form with z representing 0 and s representing the successor function.

$$\begin{array}{l}
\mathsf{nat}(\mathsf{z})\\
\forall n.\, \mathsf{nat}(n) \supset \mathsf{nat}(\mathsf{s}(n))\\[1ex]
\forall n.\, \mathsf{nat}(n) \supset \mathsf{variable}(\mathsf{var}(n))\\[1ex]
\forall t.\, \mathsf{variable}(t) \supset \mathsf{term}(t)\\[1ex]
\forall A.\, \forall B.\, \mathsf{formula}(A) \wedge \mathsf{formula}(B) \supset \mathsf{formula}(\mathsf{imp}(A,B))\\
\forall A.\, \mathsf{formula}(A) \supset \mathsf{formula}(\mathsf{not}(A))\\
\forall x.\, \forall A.\, \mathsf{variable}(x) \wedge \mathsf{formula}(A) \supset \mathsf{formula}(\mathsf{forall}(x,A))
\end{array}$$

We have to add clauses for particular function and predicate symbols. For example, if an equality predicate eq^2 is available in the object logic, we add the clause

$$\forall x.\, \forall y.\, \mathsf{term}(x) \land \mathsf{term}(y) \supset \mathsf{formula}(\mathsf{eq}(x, y))$$

Arities of the function symbols and predicates are thus built into the representation. A drawback with this and related first-order, uni-typed methods is that we have to *prove* $\mathsf{formula}(t)$ to verify that t represents a formula of the object language; it is an *external* rather than an *internal* property of the representation. More precisely, if we denote the theory above by F, then we have the following representation theorem.

2.1. THEOREM (Adequacy).

1. $F \Vdash^H \mathsf{variable}(t')$ *iff* $t' = \ulcorner x_n \urcorner$ *for a variable* x_n.
2. $F \Vdash^H \mathsf{term}(t')$ *iff* $t' = \ulcorner t \urcorner$ *for a term* t.
3. $F \Vdash^H \mathsf{formula}(t')$ *iff* $t' = \ulcorner A \urcorner$ *for a formula* A.

PROOF. In one direction this follows by an easy induction on n and the structure of t and A.

In the other direction we need a deep semantic or proof-theoretic understanding of Horn logic. For example, we use the structure of the least Herbrand model, or we can take advantage of the fact that a Horn theory inductively defines its atomic predicates. □

Adequacy theorems play a critical role in logical frameworks. They guarantee that we can translate expressions from the object language to objects in the meta-language, compute with them, and then interpret the results back in the object language. This will be particularly important when we consider the adequacy of the encoding of inference rules (Theorem 3.1) and deductions (Theorem 3.2), because they ensure that formal reasoning in the logical framework is correct with respect to the object logic under consideration. Generally, we would like the representation function to be a bijection, but this is not always necessary as long as we can translate safely in both directions.

For the particular adequacy theorem above it is irrelevant whether the propositions of the meta-logic are interpreted classically or intuitionistically, since classical and intuitionistic provability coincide on Horn clauses. We can also view a fixed set of Horn clauses as an inductive definition of the atomic predicates involved. In our example, the predicates nat, $\mathsf{variable}$, term, and $\mathsf{formula}$ are all inductively defined by the clauses given above. The fact that Horn clauses allow such diverse interpretations is one reason why they constitute a stable and frequently used basis for logical frameworks.

The first-order representation above does not support the variable convention: renaming of bound variables must be implemented explicitly. For example, the representations of $\forall x_1.\, p(x_1)$ and $\forall x_3.\, p(x_3)$ are not identified in the meta-language. Instead we can define a binary predicate id such that $\mathsf{id}(A_1, A_2)$ holds

iff A_1 and A_2 represent formulas which differ only in the names of their bound variables. The technique of *de Bruijn indices* [de Bruijn 1972] eliminates this shortcoming without requiring a change in the expressive power of the meta-language. There, a variable is represented by a natural number n, which indicates that the variable is bound by the nth enclosing abstraction. For example, $\forall x_1.\, \forall x_5.\, p(x_5) \supset p(x_1)$ and all alphabetic variants of it would be represented as $\mathsf{forall}\,(\mathsf{forall}\,(\mathsf{imp}(\mathsf{p}(\mathsf{var}(1)), \mathsf{p}(\mathsf{var}(2)))))$. De Bruijn indices have been employed as the basic representation for many implementation and verification efforts for deductive systems (see, for example, [de Bruijn 1972, Shankar 1988]).

2.2. *Simply-typed representation*

A standard method for transforming an *external* validity condition (given here by a Horn theory) into an *internal* property of the representation is to introduce types. By designing the type system so that type checking is decidable, we turn a dynamic property into a static property. 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. This idea can be directly applied to the representation in the previous section if we also introduce a type constant for variables. We can improve upon this by enriching the representation language to include higher-order terms, which leads us to the simply-typed λ-calculus, $\lambda^{\to}$. We briefly summarize it here; for more complete discussion, see Section 6.

$$\begin{array}{lll} \text{Types} & A & ::= \; a \mid A_1 \to A_2 \\ \text{Objects} & M & ::= \; c \mid x \mid \lambda x{:}A.\, M \mid M_1\, M_2 \end{array}$$

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 $[M/x]N$ for 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.

Using the simply-typed λ-calculus $\lambda^{\to}$ as a representation language requires us to distinguish between arbitrary well-typed objects and *canonical forms*. Canonical forms directly represent object-language entities, while the meaning of arbitrary well-typed objects is computed by converting them to canonical form. This is similar to most programming languages where values represent data and the meaning of an expression is determined by evaluation. This point of view leads to the following principal judgments. They are parametrized by a signature Σ that declares type and object constants and a context Γ that declares the type of variables free in M

and M'.

$$
\begin{array}{ll}
\Gamma \vdash_\Sigma M : A & M \text{ is an object of type } A \\
\Gamma \vdash_\Sigma M' \Uparrow A & M' \text{ is a canonical object of type } A \\
\Gamma \vdash_\Sigma M \Uparrow M' : A & M \text{ has canonical form } M' \text{ at type } A
\end{array}
$$

The formal definition of the language and these judgments can be found in Section 6. The appropriate notion for canonical forms are long $\beta\eta$-normal forms, that is, β-reduced and η-expanded objects. Given a syntactic category in the object language and its representation type A, canonical forms of type A are in bijective correspondence with object-language expressions in the appropriate syntactic category (see Theorem 2.2 and the subsequent discussion). Since every valid object has a unique type and canonical form (see Theorem 6.1), the meaning of an arbitrary valid object is unambiguously determined.

Two objects are *definitionally equal* if they have the same canonical form.

$$\Gamma \vdash_\Sigma M \equiv N : A \qquad M \text{ is definitionally equal to } N \text{ at type } A.$$

This is equivalent to stipulating that two objects are definitionally equal if they can be transformed into each other by β- and η-conversion. Since canonical forms depend on types, definitional equality also depends on types, although we sometimes abbreviate it as $M \equiv N$. Formulations of typed λ-calculi as the foundation for functional programming normally do not include η-conversion, since it does not preserve observational equivalence under the usual operational semantics. For example, the Pure Type Systems reviewed in [Barendregt and Geuvers 2001] (Chapter 18 of this Handbook) typically do not include η-conversion.

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

$$
\begin{array}{lll}
\mathsf{i} & : & \mathsf{type} \\
\mathsf{o} & : & \mathsf{type}
\end{array}
$$

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.

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

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.

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

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, see [Andrews 2001] (Chapter 15 of this Handbook), and Martin-Löf's system of arities [Nordström, Petersson and Smith 1990]. In programming environments this was proposed by Huet and Lang [1978] and developed further by Pfenning and Elliott [1988].

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 $(\lambda x{:}\mathsf{i}.\, \ulcorner A \urcorner)$ of type $\mathsf{i} \to \mathsf{o}$. What does it mean to apply this function? This question leads to the concept of *compositionality*, a crucial property of higher-order abstract syntax. First we note that

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

since β-conversion is an admissible rule for definitional equality. We can further prove (by a simple induction) that

$$[\ulcorner t \urcorner / x] \ulcorner A \urcorner = \ulcorner [t/x] A \urcorner.$$

Here, substitution (both at the object and meta-level) 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 β-reduction 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. In logic programming, this is called the *open-world assumption.* However, the definition also has an inductive character in the sense that the validity judgment of the meta-language ($\lambda^{\rightarrow}$, 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 canonical 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.

2.2. THEOREM (Adequacy).

1. *We have*

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

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

2. *We have*

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

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*

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

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 $M \Uparrow \mathsf{i}$ and $M \Uparrow \mathsf{o}$. 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. □

An important aspect of this theorem is that it establishes a bijection between canonical forms of a given type (i and o) and the object-language entities we are trying to represent (terms and formulas, respectively). It is clear that not every well-typed object of type i or o lies in the image of the representation function. The next two examples show that canonical forms and not just β-normal forms are actually required. We assume we have one unary predicate p and a corresponding constant $\mathsf{p}{:}\mathsf{i} \rightarrow \mathsf{o}$.

$$\vdash \mathsf{forall}\,(\lambda x{:}\mathsf{i}.\,((\lambda q{:}\mathsf{o}.\,q)\;(\mathsf{p}\,x))) : \mathsf{o}$$
$$\vdash \mathsf{forall}\,\mathsf{p} : \mathsf{o}$$

Both of these object have type o but are not in the image of the representation function $\ulcorner - \urcorner$. Their meaning can be determined by conversion to canonical form.

We calculate

$$\vdash \mathsf{forall}\,(\lambda x{:}\mathsf{i}.\,((\lambda q{:}\mathsf{o}.\,q)\;(\mathsf{p}\,x))) \Uparrow \mathsf{forall}\,(\lambda x{:}\mathsf{i}.\,\mathsf{p}\,x) : \mathsf{o}$$
$$\vdash \mathsf{forall}\,\mathsf{p} \Uparrow \mathsf{forall}\,(\lambda x{:}\mathsf{i}.\,\mathsf{p}\,x) : \mathsf{o}$$

and thus both objects represent $\forall x.\,P(x)$ (or an alphabetic variant, of course). Similar examples exist for the representation of derivations in Section 3. This shows that canonical forms play the role of *observable values* in a functional language, and conversion to canonical form the role of evaluation. A simple β-normal form would not be sufficient, as the second example illustrates.

We summarize the concepts and techniques introduced in this section. We noted the tension between *external* and *internal* validity of representations. The former arises if we write a general (logical) specification that allows us to prove that meta-language objects represent well-formed object-language expressions. The latter arises from a *typed* meta-language where well-typed meta-language objects correspond to well-formed expressions of the object language. Validity of internal representations are decidable by design, while this issue has to be reexamined in each case for external validity.

A central issue in the representation of syntax is the treatment of variables. An encoding where variables are represented by constants in the meta-language is awkward and requires a significant machinery to handle the frequently required operations of bound variable renaming and substitution. The more advanced technique of *de Bruijn indices* represents occurrences of bound variables by pointers to their binding occurrence, drastically simplifying many operations. Substitution must still be axiomatized explicitly. 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 β-reduction, both of which preserve definitional equality.

Languages such as the formulas of first-order logic are essentially *open-ended* in the sense that we may obtain specific theories by making a commitment to a particular set of function and predicate symbols. On the other hand they are also *inductive* in the sense that in order to prove a meta-theoretic property, we may need to proceed by induction over the structure of formulas, which is only possible if we know that we are considering all possible cases. The compositionality of the representation function is a simple example of such an inductive proof. This tension is reflected in the simply-typed λ-calculus as a representation language. On the one hand, it is open-ended in the sense that we can always declare new constants without invalidating any prior typing or equality judgments. On the other hand, the canonical objects constructed over a fixed signature are inductively defined, since the meta-language has an inductive definition. Some frameworks, such as FS_0 [Feferman 1988, Matthews et al. 1993] or ALF [Nordström 1993] make the inductive nature of these definitions explicit, at the price of giving up higher-order abstract syntax. On the other hand one can then reason internally about properties of deductive systems by induction. We will come back to inductive meta-reasoning in Section 5.

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. There is the *axiomatic style* (originated by Frege [1879] and in its modern form by Hilbert and Bernays [1934]) where a logical system is given by axioms and a minimal number of inference rules. Gentzen [1935] developed *natural deduction* in which the meaning of each logical symbol is explained by means of its introduction and elimination rules. Natural deductions were developed to model mathematical reasoning practices more closely than axiomatic derivations while still remaining completely formal. Gentzen also introduced *sequent calculi* in which certain properties of derivations (such as the subformula property) are explicit. Sequent calculi form the basis of many proof search procedures today. Yet another style of presentation is based on category theory [Lambek and Scott 1986].

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 hereditary Harrop formulas and the LF type theory, so we discuss the problem of representing natural deduction first. We return to axiomatic systems in Section 3.5. Other systems, including sequent calculi, can also be directly encoded [Pfenning 1995, 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 [Martin-Löf 1985*a*]. We will generally interpret the notions as formal and syntactic, 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

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

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 $[\mathcal{E}/u]\mathcal{D} :: J_2$. 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

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

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 correspond 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.

$$\begin{array}{c}\mathcal{D}\\ \overset{N}{\vdash} A\end{array} \implies_R \begin{array}{c}\mathcal{D}'\\ \overset{N}{\vdash} A\end{array} \qquad \mathcal{D} \text{ locally reduces to } \mathcal{D}'$$

$$\begin{array}{c}\mathcal{D}\\ \overset{N}{\vdash} A\end{array} \implies_E \begin{array}{c}\mathcal{D}'\\ \overset{N}{\vdash} A\end{array} \qquad \mathcal{D} \text{ locally expands to } \mathcal{D}'$$

In the framework of *partial inductive definitions* [Hallnäs 1991] when used as a meta-logic [Hallnäs 1987, Schroeder-Heister 1991, Eriksson 1992, Eriksson 1993*b*, Eriksson 1993*a*, Eriksson 1994] the specification of introduction rules for a connective automatically leads to the proper elimination rules by virtue of general properties of the framework. We do not presuppose such a mechanism, but explicitly describe both introduction and elimination rules. In the spirit of orthogonality, we proceed connective by connective, discussing introduction and elimination rules and local reductions and expansions.

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

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

The hypothetical derivation describes a construction by which we can transform a derivation of $\overset{N}{\vdash} A$ into a derivation of $\overset{N}{\vdash} B$. This is accomplished by substituting

the derivation of $\overset{N}{\vdash} A$ for every use of the hypothesis $\overset{N}{\vdash} A$ labelled u in the derivation of $\overset{N}{\vdash} B$. The elimination rule expresses just that: if we have a derivation of $\overset{N}{\vdash} A \supset B$ and also a derivation of $\overset{N}{\vdash} A$, then we can obtain a derivation of $\overset{N}{\vdash} B$.

$$\dfrac{\overset{N}{\vdash} A \supset B \qquad \overset{N}{\vdash} A}{\overset{N}{\vdash} B}\supset\!\mathrm{E}$$

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

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

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.

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

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

Negation. In order to derive $\overset{N}{\vdash} \neg A$ we assume $\overset{N}{\vdash} A$ 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 $\overset{N}{\vdash} A$ and tries to derive $\overset{N}{\vdash} B$ and $\overset{N}{\vdash} \neg B$ 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 $\overset{N}{\vdash} \neg A$ should be derivable if we can conclude $\overset{N}{\vdash} p$ for any formula p from the assumption $\overset{N}{\vdash} A$. In other words, the deduction of the premise is hypothetical in the assumption $\overset{N}{\vdash} A$ and parametric in the formula p.

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

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.

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

The local expansion is also similar to that for implication.

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

Universal quantification. Under which circumstances should we be able to derive $\stackrel{N}{\vdash} \forall x.\, A$? This clearly depends on the domain of quantification. For example, if we know that x ranges over the natural numbers, then we can conclude $\stackrel{N}{\vdash} \forall x.\, A$ if we can derive $\stackrel{N}{\vdash} [0/x]A$, $\stackrel{N}{\vdash} [1/x]A$, 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 $\stackrel{N}{\vdash} \forall x.\, A$ should be derivable if $\stackrel{N}{\vdash} [a/x]A$ is derivable for an arbitrary new parameter a. Conversely, if we know $\stackrel{N}{\vdash} \forall x.\, A$, we know that $\stackrel{N}{\vdash} [t/x]A$ for any term t.

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

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 $\stackrel{N}{\vdash} [a/x]A$ is derivable for an arbitrary a, we can conclude that $\stackrel{N}{\vdash} [t/x]A$ should be derivable for any term t. Thus we have the reduction

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

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 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 $[t/a][t/x]A$. Similarly, we would change the assumptions if a occurred free in any of the undischarged hypotheses. This might render a larger derivation incorrect. As an example, consider the judgment $\stackrel{N}{\vdash} \forall x.\, \forall y.\, p(x) \supset p(y)$ which should clearly not be derivable for an arbitrary predicate p. The following is *not* a deduction of this judgment.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\stackrel{N}{\vdash} P(a)}\,u}{\stackrel{N}{\vdash} \forall x.\, P(x)}\forall\mathrm{I}^a?}{\stackrel{N}{\vdash} P(b)}\forall\mathrm{E}}{\stackrel{N}{\vdash} P(a) \supset P(b)}\supset\mathrm{I}^u}{\stackrel{N}{\vdash} \forall y.\, P(a) \supset P(y)}\forall\mathrm{I}^b}{\stackrel{N}{\vdash} \forall x.\, \forall y.\, P(x) \supset P(y)}\forall\mathrm{I}^a$$

The flaw is at the inference marked with "?," where a is free in the assumption u. Applying a local proof reduction to the (incorrect) $\forall$I inference followed by $\forall$E leads to the assumption $[b/a]P(a)$ which is equal to $P(b)$. The resulting derivation

$$\dfrac{\dfrac{\dfrac{\overline{\stackrel{N}{\vdash} P(b)}\,u}{\stackrel{N}{\vdash} P(a) \supset P(b)}\supset\mathrm{I}^u}{\stackrel{N}{\vdash} \forall y.\, P(a) \supset P(y)}\forall\mathrm{I}^b}{\stackrel{N}{\vdash} \forall x.\, \forall y.\, P(x) \supset P(y)}\forall\mathrm{I}^a$$

is once again incorrect since the hypothesis labelled u should be $P(a)$, not $P(b)$.

The local expansion just introduces and immediately discharges the parameter.

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

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.

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

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.

Here is a simple example of a natural deduction showing that $\overset{N}{\vdash} A \supset \neg\neg A$ is derivable in intuitionistic logic. We attempt to show the process by which such a deduction may have been generated, as well as the final deduction. The three vertical dots indicate a gap in the derivation we are trying to construct, with hypotheses shown above and the desired conclusion below the gap. A trace of this process when the search is carried out in a logical framework is given in Section 4.4.

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

$$\rightsquigarrow\quad \begin{array}{c}\overline{\overset{N}{\vdash} A}^{\,u} \quad \overline{\overset{N}{\vdash} \neg A}^{\,w}\\ \vdots \\ \dfrac{\dfrac{\overset{N}{\vdash} p}{\overset{N}{\vdash} \neg\neg A}\,\neg\mathrm{I}^{p,w}}{\overset{N}{\vdash} A \supset \neg\neg A}\,{\supset}\mathrm{I}^u\end{array} \quad\rightsquigarrow\quad \dfrac{\dfrac{\dfrac{\overline{\overset{N}{\vdash} \neg A}^{\,w} \quad \overline{\overset{N}{\vdash} A}^{\,u}}{\overset{N}{\vdash} p}\,\neg\mathrm{E}}{\overset{N}{\vdash} \neg\neg A}\,\neg\mathrm{I}^{p,w}}{\overset{N}{\vdash} A \supset \neg\neg A}\,{\supset}\mathrm{I}^u$$

The symbol A in this deduction stand for an arbitrary formula; we can thus view the derivation above as parametric in A. In other words, every instance of this derivation (replacing A by an arbitrary formula) is a valid derivation.

Below is a summary of the rules of intuitionistic natural deduction. The use of hypotheses is implicit in this formulation, using our understanding of hypothetical judgments.

Introduction Rules	Elimination Rules
$\begin{array}{c}\overline{\overset{N}{\vdash} A}^{\,u}\\ \vdots \\ \dfrac{\overset{N}{\vdash} B}{\overset{N}{\vdash} A \supset B}\,{\supset}\mathrm{I}^u\end{array}$	$\dfrac{\overset{N}{\vdash} A \supset B \qquad \overset{N}{\vdash} A}{\overset{N}{\vdash} B}\,{\supset}\mathrm{E}$
$\begin{array}{c}\overline{\overset{N}{\vdash} A}^{\,u}\\ \vdots \\ \dfrac{\overset{N}{\vdash} p}{\overset{N}{\vdash} \neg A}\,\neg\mathrm{I}^{p,u}\end{array}$	$\dfrac{\overset{N}{\vdash} \neg A \qquad \overset{N}{\vdash} A}{\overset{N}{\vdash} C}\,\neg\mathrm{E}$
$\dfrac{\overset{N}{\vdash} [a/x]A}{\overset{N}{\vdash} \forall x.\, A}\,\forall\mathrm{I}^a$	$\dfrac{\overset{N}{\vdash} \forall x.\, A}{\overset{N}{\vdash} [t/x]A}\,\forall\mathrm{E}$

3.3. Representing derivability

There are several approaches to the representation of natural deductions in logical frameworks. We can introduce a predicate nd such that $\mathsf{nd}(\ulcorner A \urcorner)$ holds in the meta-logic if and only if $\overset{N}{\vdash} A$ has a derivation. This does not require an explicit representation of natural deductions as objects in the meta-language. Another possibility is to introduce an explicit representation for natural deductions and encode the property "$\mathcal{D}$ *is a deduction of* $\overset{N}{\vdash} A$".

We first consider the encoding of derivability via axioms in a meta-logic. In order to take advantage of higher-order abstract syntax in the representation, we need to go beyond Horn clauses as introduced in Section 2.1. An appropriate language is the language of *hereditary Harrop formulas* [Miller et al. 1991] which form the basis both of the logic programming language λProlog [λ*Prolog* 1997] and the generic theorem prover Isabelle [Paulson 1994]. Variations of this approach to encoding derivability have been devised by Paulson [1986] and Felty and Miller [1988, 1989]. Quantifiers in the meta-logic have type labels and range over simply-typed λ-terms. Since it is unnecessary for our purposes, we exclude quantification over formulas in the meta-logic and omit some logical connectives that are easily definable. The meta-variable A ranges here over simple types as in Section 6 and should not be confused with the formulas of first-order logic in the preceding section.

$$\text{Hereditary Harrop formulas} \quad H \quad ::= \quad P \mid \top \mid H_1 \wedge H_2 \mid H_1 \supset H_2 \mid \forall x{:}A.\, H$$

There are two important differences to Horn logic: the addition of types so that quantifiers now range over simply-typed λ-terms, and the generalization which allows the body of clauses to contain implications and universal quantifications (so-called *embedded implication* and *embedded universal quantification*). On this fragment classical and intuitionistic logic diverge, so it is crucial that the meta-logic is intuitionistic. A theory T is a collection of closed hereditary Harrop formulas.

$$T \vdash^{HH} H \qquad \text{theory } T \text{ intuitionistically entails proposition } H$$

The extension to allow embedded implications also means that theories consisting of hereditary Harrop formulas no longer constitute inductive definitions the way Horn clauses do.

Derivability by natural deductions is represented by a predicate nd on representations of formulas, that is, meta-level terms of type o. The inference rules are then translated into meta-level axioms concerning the predicate nd. For example, the rule $\supset$E is implemented by

$$\forall A{:}\mathsf{o}.\, \forall B{:}\mathsf{o}.\, (\mathsf{nd}\,(\mathsf{imp}\, A\, B) \;\wedge \mathsf{nd}\, A) \supset \mathsf{nd}\, B$$

In order to represent hypothetical judgments we take advantage of embedded implication. This is correct only because the meta-logic is intuitionistic and a complete strategy for proving a formula $H_1 \supset H_2$ is to prove H_2 under assumption H_1. Using this fact, one can prove that the following axiom is an adequate representation of the $\supset$I rule.

$$\forall A{:}\mathsf{o}.\, \forall B{:}\mathsf{o}.\, (\mathsf{nd}\, A \supset \mathsf{nd}\, B) \supset \mathsf{nd}\,(\mathsf{imp}\, A\, B)$$

For parametric judgments we can use a similar encoding with embedded universal quantification. We state the remaining rules here for completeness; the same idea is employed in the type-theoretic treatment in Section 3.4 and explained there in detail.

$$\begin{array}{l}
\forall A{:}\mathsf{o}.\, (\forall p{:}\mathsf{o}.\, \mathsf{nd}(A) \supset \mathsf{nd}(p)) \supset \mathsf{nd}(\neg A) \\
\forall A{:}\mathsf{o}.\, \mathsf{nd}(\neg A) \supset \forall C{:}\mathsf{o}.\, (\mathsf{nd}(A) \supset \mathsf{nd}(C)) \\
\forall A{:}\mathsf{i} \to \mathsf{o}.\, (\forall x{:}\mathsf{i}.\, \mathsf{nd}\,(A\, x)) \supset \mathsf{nd}\,(\mathsf{forall}\,(\lambda x{:}\mathsf{i}.\, A\, x)) \\
\forall A{:}\mathsf{i} \to \mathsf{o}.\, \mathsf{nd}\,(\mathsf{forall}\,(\lambda x{:}\mathsf{i}.\, A\, x)) \supset (\forall x{:}\mathsf{i}.\, \mathsf{nd}\,(A\, x))
\end{array}$$

We summarize the representation principle in the phrase *judgments-as-propositions*: judgments of the object language (e.g., $\vdash^{N} A$) are represented by a proposition in the meta-logic (e.g., $\mathsf{nd}(\ulcorner A \urcorner)$). The adequacy theorem of this representation is rather direct. We refer to the theory consisting of the type declarations and the six axioms above as ND.

3.1. THEOREM (Adequacy).

$$ND \vdash^{HH} \mathsf{nd}(\ulcorner A \urcorner) \quad \textit{iff} \quad \vdash^{N} A$$

In order to prove this theorem, we need to generalize it to account for hypothetical judgments. One possible form employs meta-level implication.

$$ND \vdash^{HH} \mathsf{nd}(\ulcorner A_1 \urcorner) \supset \cdots \supset \mathsf{nd}(\ulcorner A_n \urcorner) \supset \mathsf{nd}(\ulcorner A \urcorner) \quad \text{iff} \quad \begin{array}{c} \overline{\vdash^{N} A_1}^{\,u_1} \;\cdots\; \overline{\vdash^{N} A_n}^{\,u_n} \\ \vdots \\ \vdash^{N} A \end{array}$$

Another form, given for the related type-theoretic interpretation in the next section, directly uses hypothetical reasoning in the meta-language.

3.4. Deductions as objects

If we have a general reasoning tool for hereditary Harrop formulas we can now reason in intuitionistic logic by using the axioms in the theory ND, and in classical logic if we assume an additional axiom modelling double negation elimination. Isabelle [Paulson 1994, Nipkow and Paulson 1992] is such a general tool. Proof search can be programmed externally by using a language of tactics and tacticals to construct derivations using these axioms and derived rules of inference. The meta-programming language in this case is ML, whose type system together with a correct implementation of hereditary Harrop formulas guarantees that only well-formed meta-derivations can be constructed. More on this style of reasoning with the aid of a logical framework implementation can be found in Section 4. As mentioned above, this is an implementation of *derivability* and explicit deductions need never be constructed. If they are maintained, they are only an internal data structure.

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 (see Section 8.2). 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 axiomatic derivations define the same theorems and exhibit translations between them (see Sections 5.2 and 5.4).

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 instead of the predicate introduced in the previous section. An inference rule is a constant function from deductions of the premises to a deduction of the conclusion. For example,

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

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.

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

One can clearly see that this representation requires an *external* validity condition since it does not carry the information about the conclusion of a derivation. For example, we have

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

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, Felty 1989]. However, it is dynamic (rather than static) and not *prima facie* decidable. Furthermore, during search external mechanisms must be put into place in order to prevent invalid deductions. This is related to the problem of *invalid tactics* in ML/LCF [Gordon, Milner and Wadsworth 1979]. Through data abstraction, tactics are guaranteed to generate only valid deductions, but the type system cannot enforce that they have the expected conclusion.

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.

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash_{\Sigma} M : \Pi x{:}A.\,B \qquad \Gamma \vdash_{\Sigma} N : A}{\Gamma \vdash_{\Sigma} M\ N : [N/x]B}\,\mathsf{app}$$

For example, given a variable $p{:}\mathsf{o}$ we have

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

where the signature Σ contains the declarations for formulas and inferences rules developed above. The counterexample $\mathsf{impi}\,(\lambda u{:}\mathsf{nd}\,A.\,\mathsf{impe}\,u\,u)$ 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.

$$\frac{\Gamma \vdash_{\Sigma} A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_{\Sigma} M : B}{\Gamma \vdash_{\Sigma} \lambda x{:}A.\,M : \Pi x{:}A.\,B}\,\mathsf{lam}$$

The variable x may now appear free in B, whereas without dependencies it could only occur free in M. 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.

Dependent types further create the need for a rule of *type conversion*. This is required, for example, in the representation of $\forall$I below. We take a brief excursion into the realm of functional programming to illustrate the nature of dependent types and the need for type conversion. Consider a type family vector indexed by a natural number representing its length. Then concatenation of vectors would have type

$$\mathsf{concat} \quad : \quad \Pi n{:}\mathsf{nat}.\,\Pi m{:}\mathsf{nat}.\,\mathsf{vector}\,n \to \mathsf{vector}\,m \to \mathsf{vector}\,(n+m).$$

Using the inference rules for application we find

$$\begin{array}{lcl} \mathsf{concat}\,2\,3\,[1,2]\,[1,3,5] & : & \mathsf{vector}(2+3),\ \text{and} \\ \mathsf{concat}\,3\,2\,[1,2,1]\,[3,5] & : & \mathsf{vector}(3+2). \end{array}$$

Since both expressions compute to the same value, namely

$$[1,2,1,3,5] \quad : \quad \mathsf{vector}(5),$$

we would expect that in a sensible type system all three expressions would have the same type. Evidently they do not, unless we identify the types $\mathsf{vector}(2+3)$, $\mathsf{vector}(3+2)$, and $\mathsf{vector}(5)$. All of them represent the type of vectors of length 5, so identifying them makes sense intuitively. In general, we add a rule of type conversion that allows us to apply definitional equalities in a type.

$$\frac{\Gamma \vdash_\Sigma M : A \qquad \Gamma \vdash_\Sigma A \equiv B : \mathsf{type}}{\Gamma \vdash_\Sigma M : B}\,\mathsf{conv}$$

The example above also shows that adding dependent types to a functional language can quickly lead to an undecidable type checking problem, since we need to compare expressions in the program language for equality (which is undecidable in general). The LF type theory contains no recursion at the level of objects and type-checking remains decidable since definitional equality remains decidable. This is an important illustration of the design principle that the framework should be as weak as possible. Adding recursion, while it may occasionally seem desirable, can easily destroy decidability of definitional equality and therefore typing. In an undecidable type system, validity of the representations for deductions then would no longer be a static, internal property.

A full complement of rules for the λ^Π type theory is given in Section 7. A version with a weaker notion of definitional equality is given in [Barendregt and Geuvers 2001] (Chapter 18 of this Handbook)

With dependent function types, we can now give a representation for natural deductions with an internal validity condition. This is summarized in Theorem 3.2 below. We first introduce a type family nd that is indexed by a formula. The LF type $\mathsf{nd}\ulcorner A\urcorner$ is intended to represent the type of natural deductions of the formula A.

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

Each inference rule is represented by an LF constant which can be thought of as a function from a derivation of the premises of the rule to a derivation of the conclusion. The constant further depends on the schematic variables that occur in the specification of the inference rule.

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.

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

The assumption A labelled by u which may be used in the derivation $\mathcal{D}$ is represented by the LF variable u which ranges over derivations of A.

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

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

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

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

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

where

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

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

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

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

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

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 occurrence 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*a*, Necula and Lee 1998*b*]. Section 8.2 has some additional brief remarks on this issue.

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

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

The judgment of the premise is parametric in p and hypothetical in u. It is thus represented as a function of two arguments, accepting both a formula p and a deduction of A.

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

The representation of negation elimination

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

leads to the following declaration

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

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

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

is an equivalent declaration.

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

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

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.

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

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

The elimination rule does not employ a hypothetical judgment.

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

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

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

We now check that

$$\ulcorner\ \dfrac{\begin{array}{c}\mathcal{D}\\ \overset{N}{\vdash} \forall x.\,A\end{array}}{\overset{N}{\vdash} [t/x]A}\forall\mathrm{E}\ \urcorner \quad : \mathsf{nd}\ \ulcorner[t/x]A\urcorner,$$

assuming that $\ulcorner\mathcal{D}\urcorner : \mathsf{nd}\ \ulcorner\forall x.\,A\urcorner$. 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.

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

The last step follows by type conversion, noting that

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

Furthermore, by the compositionality of the representation we have

$$[\ulcorner t\urcorner/x]\ulcorner A\urcorner = \ulcorner[t/x]A\urcorner$$

which yields the desired

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

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.

3.2. THEOREM (Adequacy).

1. *If $\mathcal{D}$ is a derivation of A from hypotheses $\stackrel{N}{\vdash} A_1, \ldots, \stackrel{N}{\vdash} A_n$ 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*

$$a_1{:}\mathsf{i}, \ldots, a_m{:}\mathsf{i}, p_1{:}\mathsf{o}, \ldots, p_k{:}\mathsf{o}, u_1{:}\mathsf{nd}\ulcorner A_1 \urcorner, \ldots, u_n{:}\mathsf{nd}\ulcorner A_n \urcorner \vdash_{ND} \ulcorner \mathcal{D} \urcorner \Uparrow \mathsf{nd}\ulcorner A \urcorner$$

2. *If*

$$a_1{:}\mathsf{i}, \ldots, a_m{:}\mathsf{i}, p_1{:}\mathsf{o}, \ldots, p_k{:}\mathsf{o}, u_1{:}\mathsf{nd}\ulcorner A_1 \urcorner, \ldots, u_n{:}\mathsf{nd}\ulcorner A_n \urcorner \vdash_{ND} M \Uparrow \mathsf{nd}\ulcorner A \urcorner$$

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.*

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

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. □

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. We make a few cosmetic changes which reflect common practice in the use of logical frameworks. The first is the use of infix and prefix notation for logical connectives. According to our conventions, implication is right associative, and negation is a prefix operator binding more tightly than implication.

```
i : type.
o : type.
imp : o → o → o.
not : o → o.
forall : (i → o) → o.
```

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 1998*b*, Pfenning and Schürmann 1998*c*], type reconstruction will issue an error message if the type of free variables is ambiguous.

```
nd : o → type.
impi : (nd A → nd B) → nd (A imp B).
impe : nd (A imp B) → nd A → nd B.
noti : (Πp:o. nd A → nd p) → nd (not A).
note : nd (not A) → (ΠC:o. nd A → nd C).
```



```
foralli : (Πa:i. nd (A a)) → nd (forall A).
foralle : nd (forall A) → (ΠT:i. nd (A T)).
```

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

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

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

```
impi (λu:nd A. impi (λv:nd B. u)) : nd (A imp (B imp A)).
```

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 M 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.

3.5. An axiomatic formulation

A second important style of deductive system is *axiomatic*: rather than explaining the meaning of quantifiers and connectives by inference rules, we use mostly axiom schemas and as few inference rules as possible. The following is the system H_1-**IQC** [Troelstra and van Dalen 1988]. It consists of the following axiom schemas, and the two rules of inference below.

$$\begin{array}{ll} \stackrel{A}{\vdash} A \supset (B \supset A) & (K) \\ \stackrel{A}{\vdash} (A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C)) & (S) \\ \stackrel{A}{\vdash} (A \supset \neg B) \supset ((A \supset B) \supset \neg A) & (N_1) \\ \stackrel{A}{\vdash} \neg A \supset (A \supset B) & (N_2) \\ \stackrel{A}{\vdash} (\forall x.\, A) \supset [t/x]A & (F_1) \\ \stackrel{A}{\vdash} (\forall x.\, (B \supset A)) \supset (B \supset \forall x.\, A) & (F_2)^* \end{array}$$

with the proviso that x must not be free in B in the rule (F_2). The two rules of inference are modus ponens MP and universal generalization UG.

$$\frac{\stackrel{A}{\vdash} A \supset B \qquad \stackrel{A}{\vdash} A}{\stackrel{A}{\vdash} B}\, MP \qquad\qquad \frac{\stackrel{A}{\vdash} [a/x]A}{\stackrel{A}{\vdash} \forall x.\, A}\, UG^a$$

The universal generalization rule carries the proviso that a must be a new parameter, that is, may not already occur in A. The representation of the propositional axioms and modus ponens is straightforward, following the ideas in the representation of natural deduction. We introduce a type family hil for axiomatic deductions, indexed by the conclusion of the derivation. In order to improve readability, we use infix notation for implication. Also, we have chosen constant names in lower case so that the presentation of the translations in Section 5.2 will be easier to read.

$\quad\mathsf{hil} : \mathsf{o} \to \mathsf{type}.$
$\quad\mathsf{k} : \mathsf{hil}\ (A\ \mathsf{imp}\ B\ \mathsf{imp}\ A).$
$\quad\mathsf{s} : \mathsf{hil}\ ((A\ \mathsf{imp}\ B\ \mathsf{imp}\ C)\ \mathsf{imp}\ (A\ \mathsf{imp}\ B)\ \mathsf{imp}\ A\ \mathsf{imp}\ C).$
$\quad\mathsf{n}_1 : \mathsf{hil}\ ((A\ \mathsf{imp}\ \mathsf{not}\ B)\ \mathsf{imp}\ (A\ \mathsf{imp}\ B)\ \mathsf{imp}\ \mathsf{not}\ A).$
$\quad\mathsf{n}_2 : \mathsf{hil}\ (\mathsf{not}\ A\ \mathsf{imp}\ A\ \mathsf{imp}\ B).$

For rule (F_1) we need to implement substitution, which is done as usual in higher-order abstract syntax by application, here of A to T.

$\quad\mathsf{f}_1 : \Pi T{:}\mathsf{i}.\ \mathsf{hil}\ (\mathsf{forall}\ (\lambda x{:}\mathsf{i}.\ A\ x)\ \mathsf{imp}\ A\ T).$

For the rule (F_2) we must capture the side-condition that x is not free in the antecedent of the implication. The following achieves this directly.

$\quad\mathsf{f}_2 : \mathsf{hil}\ (\mathsf{forall}\ (\lambda x{:}\mathsf{i}.\ B\ \mathsf{imp}\ A\ x)\ \mathsf{imp}\ B\ \mathsf{imp}\ \mathsf{forall}\ (\lambda x{:}\mathsf{i}.\ A\ x)).$

Since substitution in the meta-language will rename bound variables to avoid variable capture, we cannot instantiate B in this declaration with an object that contains a free occurrence of x (x would be renamed). Thus, using higher-order abstract syntax, one can concisely represent simple variable occurrence conditions. The rules of inference are isomorphic to ones we have seen for natural deduction.

$\quad\mathsf{mp} : \mathsf{hil}\ (A\ \mathsf{imp}\ B) \to \mathsf{hil}\ A \to \mathsf{hil}\ B.$
$\quad\mathsf{ug} : (\Pi a{:}\mathsf{i}.\ \mathsf{hil}\ (A\ a)) \to \mathsf{hil}\ (\mathsf{forall}\ (\lambda x{:}\mathsf{i}.\ A\ x)).$

The adequacy theorem for axiomatic derivations is straightforward and left to the reader.

3.6. Higher-level judgments

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

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

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.

$\quad\Longrightarrow_R \;:\; \Pi A{:}\mathsf{o}.\ \mathsf{nd}\ A \to \mathsf{nd}\ A \to \mathsf{type}.$

As in the representation of inference rules in Sections 3.4 and 3.5, we omit the explicit quantifier on A and determine A from context.

$\quad\Longrightarrow_R : \mathsf{nd}\ A \to \mathsf{nd}\ A \to \mathsf{type}.$

We show the representation of the reduction rules for each connective in turn, writing $\Longrightarrow_R$ as an infix constant.

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

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

The representation of the left-hand side is

$\quad\mathsf{impe}\ (\mathsf{impi}\ (\lambda u{:}\mathsf{nd}\ A.\ D\ u))\ E$

where $E = \ulcorner\mathcal{E}\urcorner : \mathsf{nd}\,A$ and $D = (\lambda u{:}\mathsf{nd}\,\ulcorner A\urcorner.\ulcorner\mathcal{D}\urcorner) : \mathsf{nd}\,A \to \mathsf{nd}\,B$. The derivation on the right-hand side can be written more succinctly as $[\mathcal{E}/u]\mathcal{D}$. Compositionality of the representation (Theorem 3.2, part 3) and β-conversion in LF yield

$$\ulcorner[\mathcal{E}/u]\mathcal{D}\urcorner = [\ulcorner\mathcal{E}\urcorner/u]\ulcorner\mathcal{D}\urcorner \equiv (\lambda u{:}\mathsf{nd}\,\ulcorner A\urcorner.\ulcorner\mathcal{D}\urcorner)\ \ulcorner\mathcal{E}\urcorner.$$

Thus the representation of the right-hand side will be definitionally equal to $D\,E$ and we can formulate the rule concisely as

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

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.

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

$\quad\mathsf{redl\_not} : \mathsf{note}\ (\mathsf{noti}\ (\lambda p{:}\mathsf{o}.\ \lambda u{:}\mathsf{nd}\ A.\ D\ p\ u))\ C\ E \Longrightarrow_R D\ C\ E.$

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.

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

In the representation we once again exploit the compositionality.

$$\ulcorner [t/a]\mathcal{D} \urcorner = [\ulcorner t \urcorner / a]\ulcorner \mathcal{D} \urcorner \equiv (\lambda a{:}\mathsf{i}.\, \ulcorner \mathcal{D} \urcorner)\ \ulcorner t \urcorner$$

This gives rise to the declaration

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

The adequacy theorem states that canonical LF objects of type $\ulcorner \mathcal{D} \urcorner \Longrightarrow_R \ulcorner \mathcal{D}' \urcorner$ 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.

$\Longrightarrow_E : \mathsf{nd}\ A \to \mathsf{nd}\ A \to \mathsf{type}.$
$\mathsf{expl\_imp} : \Pi D{:}\mathsf{nd}\ (A\ \mathsf{imp}\ B).\, D \Longrightarrow_E \mathsf{impi}\ (\lambda u{:}\mathsf{nd}\ A.\,\mathsf{impe}\ D\ u).$
$\mathsf{expl\_not} : \Pi D{:}\mathsf{nd}\ (\mathsf{not}\ A).\, D \Longrightarrow_E \mathsf{noti}\ (\lambda p{:}\mathsf{o}.\lambda u{:}\mathsf{nd}\ A.\,\mathsf{note}\ D\ p\ u).$
$\mathsf{expl\_forall} : \Pi D{:}\mathsf{nd}\ (\mathsf{forall}\ (\lambda x{:}\mathsf{i}.\, A\ x)).\, D \Longrightarrow_E \mathsf{foralli}\ (\lambda a{:}\mathsf{i}.\,\mathsf{foralle}\ D\ a).$

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 β-reduction at the meta-level, taking advantage of the compositionality of the representation. Further examples of higher-level judgments can be found in Section 5.

4. Meta-programming and proof search

An important motivation underlying the development of logical frameworks is to factor the effort required to build a theorem proving environment for specific logics. The idea is to build one generic environment for deriving judgments in the logical framework and use this for particular logical systems whose judgments are specified in the framework. Each logic is still likely to require a significant amount of development, but the goal is to reduce this effort as much as possible. Furthermore, by offering a high-level notation for the judgments of an object logic, one can increase the confidence in the correctness of an implementation, especially if the framework offers a notation for derivations independent of proof search. The practical evidence

gathered through many experiments with Isabelle in a variety of logics indicates that this is indeed feasible and fruitful.

This raises two related questions: which are the common concepts in theorem proving shared among different logics, and how do we perform search in the logical framework? We concentrate on the latter question in the hope that the almost universal applicability of the ideas becomes apparent.

4.1. Sequent calculus

Many forms of proof search are based on sequent calculi. A *sequent* generally has the form $\mathcal{J} \Longrightarrow J$ where $\mathcal{J}$ is a context of available labelled hypotheses $u_1 :: J_1, \ldots, u_n :: J_n$ and J is the judgment we are trying to derive. This is just a less cumbersome notation for hypothetical judgments as introduced in Section 3.1. We refer to each J_i as an *antecedent* and J as the *succedent* of the sequent.

Fully automatic theorem proving for practically interesting logics is rarely feasible, so framework implementations such as Isabelle are based on partially automated search. In this case, it is most intuitive to think of the construction of a derivation as proceeding bottom-up, where a sequent $\mathcal{J} \Longrightarrow J$ represents the goal of deriving J from $\mathcal{J}$. We describe the possible goal reductions in the form of inference rules for the sequent judgment. Since this view of search is a shared feature between many different logics, it is natural to base the generic search in the logical framework on the same principle, thereby directly supporting this view for a variety of object logics. The use of sequents for the top-down construction of derivations is the basis of the inverse method discussed in [Degtyarev and Voronkov 2001*b*] (Chapter 4 of this Handbook).

We describe here a sequent calculus for LF. A substantially similar and slightly simpler presentation can be given for hereditary Harrop formulas and related logical frameworks. The formulation below is based on work by Pym and Wallen [1990, 1991]. The presentation of LF motivated in Section 3.4 and summarized in Section 7 is highly economical in that the simple function type $A \to B$ is considered an abbreviation for a dependent function type $\Pi x{:}A.\, B$ where x does not occur in B. During search, however, these two are treated differently: $A \to B$ behaves like an implication, while $\Pi x{:}A.\, B$ behaves like a universal quantifier. Already in the description of our representation technique we have informally distinguished between them: $A \to B$ corresponded to a hypothetical judgment while $\Pi x{:}A.\, B$ corresponded to a parametric judgment. Our sequents have the form

$$\Gamma \overset{LF}{\Longrightarrow} M : A$$

where Γ is a context of parameter declarations and hypotheses and M is a proof term for A. During search we think of Γ and A as given, while M is filled in when a proof succeeds. We fix a signature Σ which encodes the expressions and inference rules of the object language under consideration and omit it from the judgment since it never changes. We maintain the following invariants:

1. $\vdash \Gamma\ Ctx$
2. $\Gamma \vdash A : \mathsf{type}$
3. $\Gamma \vdash M : A$

We use h to range over either a constant c declared in Σ or variable declared in Γ. We have initial sequents and so-called *right* and *left* rules for each type constructor ($\to$ and Π).

Initial sequents. We have solved a goal if a hypothesis matches the succedent, modulo definitional equality.

$$\frac{h{:}A' \text{ in } \Sigma \text{ or } \Gamma \qquad \Gamma \vdash A' \equiv A : \mathsf{type}}{\Gamma \overset{LF}{\Longrightarrow} h : A}\,\mathsf{init}$$

Hypothetical judgments. To derive the representation $A \to B$ of a hypothetical judgment, we simply introduce a hypothesis A with a new label u.

$$\frac{\Gamma, u{:}A \overset{LF}{\Longrightarrow} M : B}{\Gamma \overset{LF}{\Longrightarrow} \lambda u{:}A.\, M : A \to B}\,{\to}R^u$$

If we have an assumption $A \to B$ we are allowed to assume B if we can derive A. The conclusion C does not change in this rule.

$$\frac{h{:}A \to B \text{ in } \Sigma \text{ or } \Gamma \qquad \Gamma \overset{LF}{\Longrightarrow} M : A \qquad \Gamma, u{:}B \overset{LF}{\Longrightarrow} N : C}{\Gamma \overset{LF}{\Longrightarrow} [(h\,M)/u]N : C}\,{\to}L^u$$

Parametric judgments. To derive the representation $\Pi x{:}A.\,B$ of a parametric judgment, we simply introduce a new parameter (for convenience also called x).

$$\frac{\Gamma, x{:}A \overset{LF}{\Longrightarrow} M : B}{\Gamma \overset{LF}{\Longrightarrow} \lambda x{:}A.\, M : \Pi x{:}A.\,B}\,\Pi R^x$$

To use a parametric assumption $\Pi x{:}A.\,B$ we instantiate x with an object of the correct type.

$$\frac{h{:}\Pi x{:}A.\,B \text{ in } \Sigma \text{ or } \Gamma \qquad \Gamma \vdash M \Uparrow A \qquad \Gamma, u{:}[M/x]B \overset{LF}{\Longrightarrow} N : C}{\Gamma \overset{LF}{\Longrightarrow} [(h\,M)/u]N : C}\,\Pi L^u$$

Note that we fall back on the ordinary typing judgment for LF to check that the substitution term M is well-typed in the appropriate context. The calculus shown above is sound and complete, as shown by Pym and Wallen [1991]. As usual, we assume a fixed valid signature Σ and that Γ is valid in Σ.

4.1. THEOREM (Properties of LF sequent calculus).

1. If $\Gamma \overset{LF}{\Longrightarrow} M : A$ then $\Gamma \vdash M : A$.

2. If $\Gamma \vdash M \Uparrow A$ then $\Gamma \overset{LF}{\Longrightarrow} M : A$.

PROOF. The first property is easy to see by induction on the sequent derivation. The second can be proved by induction on the definition of canonical forms, after appropriate generalization for atomic forms (defined in Section 7). □

We can sharpen this theorem if we restrict initial sequents to atomic types P. In that case $\Gamma \overset{LF}{\Longrightarrow} M : A$ implies that $\Gamma \vdash M \Uparrow A$ (see [Pinto and Dyckhoff 1998]). The additional rule of Cut which is sometimes allowed in sequent calculi plays a special role. It corresponds to the introduction of a lemma during proof search, which is very difficult to automate. Its discussion is left to Section 4.5.

When constructing a sequent derivation upwards from the conclusion, one is confronted with a variety of choices. In particular, we have to decide which rule to apply and, for the left rules, which hypothesis to use. Usually one takes advantage of additional properties of the logic to eliminate some of the choices. For example, in the sequent calculus for LF the conclusion of $\to R$ is derivable if and only if the premise is derivable. Therefore it is always safe to apply this rule when the succedent has the form $A \to B$. Implementations of logical frameworks take advantage of such inversion properties to eliminate non-determinism in search. However, some choices clearly will always remain—they have to be addressed either via user interaction or some form of meta-programming. This is the topic of the next section.

4.2. Tactics and tacticals

In this section we address the question which choices arise during search within a sequent calculus, and how the non-determinism inherent in these choices can be resolved. We assume a meta-level control structure of so-called *tactics* and *tacticals*. As a first approximation, a tactic transforms a partial proof structure with some unproven leaf sequents to another, while a tactical is a (higher-order) function to combine tactics to form more complex tactics. At the top level, the user can choose which tactic to apply, and which unproven sequent to apply it to. We analyze the structure of tactics and tacticals in more detail when discussing the kind of choices they have to resolve.

Tactics and tacticals arose out of the LCF theorem proving effort [Gordon et al. 1979, Paulson 1983] and are used in such diverse systems as HOL [Gordon and Melham 1993], Nuprl [*Nuprl* 1999, Constable et al. 1986], Coq [*Coq* 1999, Paulin-Mohring 1993], Isabelle [Paulson 1994, Paulson 1994], and λProlog [*λProlog* 1997, Nadathur and Miller 1988, Felty 1993]. In all but λProlog, they are programmed in ML which was originally developed to support theorem proving for LCF. Correctness for tactics is ensured dynamically through *data abstraction.* The basic idea is that at the core of the implementation is an abstract type of Theorem with constructors which implement and check the correct application of the primitive rules of inference for a judgment. Since the type is abstract, only the given rules can be used, thereby reducing the correctness problem for a complex theorem proving environment to the correctness of the implementation of the basic inference rules.

In a logical framework with dependent types the correctness of deductions may instead be enforced by type-checking alone, as we have seen in Section 3.4. We therefore skip more detailed discussion of the validation of tactics and consider how they deal with choices that arise during search in a sequent calculus. In the ELAN logical framework [*ELAN* 1998, Borovanský et al. 1998] the strategy language has independent status, rather than being embedded in a general-purpose functional language such as ML. Besides individual tactic combinators to address various aspects of search, tactic languages provide general mechanisms for composition of tactics and iteration or recursion.

Conjunctive choice. A conjunctive choice arises when a sequent rule has several premises. Each of these premises must be derived to derive the conclusion. The $\rightarrow L^u$ rule has this character: to derive the judgment C we derive A, and also C under the additional hypothesis B. A tactic can choose any unproven leaf from a partial proof structure to work on, usually the leftmost pending sequent. Tactic languages provide a tactical MAP such that $\mathsf{MAP}\ \ t$ is a tactic which applies t to all pending sequents in turn. In an interactive setting the user can navigate between unproven sequents.

Disjunctive choice. A disjunctive choice arises when there are several rules which could be applied, or several different ways in which a particular rule might be applied. For example, in the $\rightarrow L^u$ rule we have to pick a hypothesis $h{:}A \rightarrow B$ from Σ or Γ when there may be several such assumptions. When tactics are employed for proof search, this is handled by *backtracking*. A tactic may apply sequent rules (from the bottom up) to reduce an unproven sequent, or it might *fail*. Failure for a tactic to apply signals that an alternative should be tried for an earlier choice. In the language of tacticals this is expressed with the ORELSE combinator. The tactic $t_1\ \ \mathsf{ORELSE}\ \ t_2$ tries to apply t_1 and returns its result if successful. If t_1 fails it tries to apply t_2 instead and returns its result. In particular, if t_2 also fails, then $t_1\ \ \mathsf{ORELSE}\ \ t_2$ fails. We refer to this as *shallow backtracking* because when t_1 succeeds the alternative t_2 will never be reconsidered. We discuss *deep backtracking* below, when we examine the interaction between disjunctive choice and meta-variables.

Universal choice. This arises, for example, in the ΠR^a rule where we have to choose a new parameter a. Since the only relevant criterion is that a is new, this does not lead to any undesirable non-determinism: any new a suffices.

Existential choice. This arises when we have to pick a term as, for example, the object M in the rule ΠL. Early implementations of tactics typically either guessed a plausible term or required the user to supply it. Since there often are an infinite number of choices, more recent implementations usually postpone a commitment until further search uncovers information about which terms might lead to a successful derivation. We achieve this postponement by using a place-holder X for M, called a *meta-variable* or *logical variable*. In order to guarantee soundness when meta-variables are instantiated we record its type A_X and the context Γ_X which contains the parameters which are allowed to occur in the instantiation term for X. The latter constraint on X replaces Skolemization as used in classical first-order theorem proving, which does not work for all object logics and would therefore be a poor choice in a logical framework.

Postponed existential choices are resolved when initial sequents are reached. Rather than check if a hypothesis matches the succedent modulo definitional equality, we have to decide if there is a way to instantiate the meta-variables in a hypothesis and the succedent so that the resulting judgments are definitionally equal. This problem is called *unification* and discussed in the Section 4.3 and in more detail in [Dowek 2001] (Chapter 16 of this Handbook). The introduction of meta-variables into search also interacts strongly with conjunctive and disjunctive choices, which we now revisit.

Conjunctive choice with meta-variables. Meta-variables may be shared among several unproven leaf sequents. Since unification instantiates these variables globally in a partial proof structure, the order in which unproven sequents are reduced is no longer irrelevant. Tactics have to be aware of this interaction, although there are no simple and general recipes.

Disjunctive choice with meta-variables. Deriving an unproven sequent often requires a commitment to a particular instantiation for meta-variables as determined by unification at the leaves. This commitment could make it impossible to derive another sequent which shares some of the meta-variables. This means that even after successfully deriving a particular sequent, we might have to reexamine the choices made during this derivation in case another sequent turns out to be unprovable. This leads to *deep backtracking* which revisits disjunctive choices even though an alternative had previously been successful.

Under the simplified functional model for tactics introduced above, a tactic returns either no result (it fails) or a single result (the new partial proof structure). Deep backtracking requires that a tactic can return a potentially unbounded number of alternatives, where zero alternatives indicate failure. This can be done by using a lazily computed sequence of alternatives which can be incrementally expanded as necessary during backtracking. The Isabelle logical framework implementation uses this technique, since its meta-programming language ML is functional. In ELAN the operator dk (for *don't know choose*) achieves this behavior.

The λProlog and Elf implementations provide an alternative by using a logic programming interpretation of the logical framework to program search. Since logic programming inherently supports logical variables, unification, and deep backtracking, significantly less machinery is needed to implement tactics (see [Felty 1993]). On the other hand, don't-care non-determinism requires additional programming or extra-logical constructs such as the cut operator "!", since the operational interpretation of logic programs is based on don't-know non-determinism. We come back to this in Section 4.4.

We use t_1 THEN t_2 to denote the sequential composition of tactics and REPEAT t for the iterator which applies t until it fails and then returns the last result. REPEAT t is an example of an *unfailing* tactic which always succeeds, though subgoals may of course remain. The interaction of possibly failing and unfailing tactics is one of the difficulties in tactic programming.

As a simple example, assume we have basic tactics Init, ArrowR, and PiR which apply the rules init, $\rightarrow R$ and ΠR, respectively. Then the tactic

```
Right* = REPEAT (ArrowR ORELSE PiR ORELSE Init)
```

repeatedly applies the right rules to a sequent until the succedent is atomic. The atomic goal is solved if it unifies with a hypothesis; otherwise it remains as a subgoal. This tactic is *safe*, that is, if the original sequent is derivable, the resulting sequent will still be derivable. Right* is safe, despite the fact that we use a committed choice tactical ORELSE, since the right rules of the sequent calculus for λ^{Π} are *invertible*: the premise is derivable if and only if the conclusion is derivable. The interaction of safe and unsafe tactics is another complicated aspect of tactic programming.

4.3. Unification and constraint simplification

As sketched above, unification is a central and indispensable mechanism in traditional first-order theorem provers and logic programming languages. It allows the search algorithm to postpone existential choices until more information becomes available about which instances may be useful. Most logical frameworks go beyond first-order terms in two ways: they employ types and they employ λ-abstraction. Consequently, first-order unification is insufficient. In this section we briefly review the aspects of higher-order unification most relevant to the practice of logical frameworks. For more information see [Dowek 2001] (Chapter 16 of this Handbook).

One can identify the simply-typed λ-calculus ($\lambda^{\rightarrow}$) as motivated in Section 2.2 as an important base language. Fortunately, definitional equality ($\beta\eta$-conversion) is decidable. On the other hand, the general unification problem is undecidable [Huet 1973] even for the second-order fragment [Goldfarb 1981], and most general unifiers may not exist. To appreciate some of the problems of higher-order unification, consider the equation

$$(\lambda x{:}\mathsf{i}.\, F\,(\mathsf{s}\,x)) = (\lambda x{:}\mathsf{i}.\, \mathsf{s}\,(F\,x))$$

where $s{:}\mathsf{i} \rightarrow \mathsf{i}$ is a constant, and F is a meta-variable we are trying to solve for. Note that F itself may not contain free occurrences of x according to the definition of capture avoiding substitution. There are infinitely many different solutions for F, namely

$$(\lambda y{:}\mathsf{i}.\, \mathsf{s}\,\ldots(\mathsf{s}\,y))$$

for any number of applications of s, including zero.

Despite the undecidability, Huet [1975] devised a practical algorithm for *higher-order pre-unification*, a form of unification which postpones certain solvable equations instead of enumerating their solutions. The resulting semi-decision procedure is non-deterministically complete, that is, if there is a unifier a less committed pre-unifier can in principle always be found. Moreover, when used to compute multiple solutions, it is guaranteed to enumerate non-redundant pre-unifiers to a given set of equations. With the addition of a modified version of the occurs-check, it coincides with first-order unification when called on first-order terms. Huet's algorithm has been used extensively in λProlog and Isabelle and generally seems to have good computational properties. Both languages must therefore manage constraints during search or execution of programs [Kirchner, Kirchner and Vittek 1993].

The practical success of Huet's algorithm seemed to be in part due to the fact that difficult, higher-order unification problems rarely arise in practice. An analysis of this observation led Miller [1991] to discover *higher-order patterns*, a sublanguage of the simply-typed λ-calculus with restricted variable occurrences. For this fragment, most general unifiers exist. In fact, the theoretical complexity of this problem is linear [Qian 1993], just as for first-order unification. Miller proposed it as the basis for a lower-level language L_λ similar to λProlog, but one where unification does not branch since only higher-order patterns are permitted as terms. An empirical study of this restriction by Michaylov and Pfenning [1992, 1993] showed that most dynamically arising unification problems lie within this fragment, while a static restriction rules out some useful programming idioms.

The Elf language therefore makes no syntactic restriction to higher-order patterns, nor does it use Huet's algorithm for higher-order unification as generalized to λ^{Π} (discovered independently by Elliott [1989, 1990] and Pym [1990, 1992]). Instead, it employs a constraint solving algorithm [Pfenning 1991*a*, Pfenning 1991*b*, Dowek, Hardin, Kirchner and Pfenning 1996] where unification problems within the decidable fragment proposed by Miller are solved directly, while all others (solvable or not) are postponed as constraints. This can drastically reduce backtracking compared to higher-order pre-unification and imposes no restrictions on variable occurrences. On the other hand, unsolvable constraints may remain until the end of the computation, in which case the answer is conditional: Each solution to the remaining constraints gives rise to a solution of the original equations, and each solution to the original equations will be an instance of the remaining constraints. In most practical applications, these somewhat weaker soundness and completeness theorems are sufficient.

4.4. Logic programming

Logic programming offers a different approach to meta-programming in a logical framework than ML or a separate strategy language. Rather than meta-programming in a language in which the logical framework itself is implemented (typically ML), we endow the logical framework with an operational interpretation in the spirit of Prolog. It should be clear that a specification of a logic under this approach does not automatically give rise to a theorem prover, but that theorem provers may be programmed in the meta-language. Two frameworks to date have pursued this approach: λProlog [*λProlog* 1997, Nadathur and Miller 1988], which gives an operational interpretation of hereditary Harrop formulas, and Elf [Pfenning and Schürmann 1998*b*, Pfenning 1994*a*], which gives an operational interpretation to λ^{Π}.

In logic programming the basic computational mechanism is proof search following a specific search strategy. Since the search strategy is fixed, the computational behavior of a program can be predicted and exploited by the programmer. This predictability comes at the price of completeness: programs may never terminate even if there is a proof. On the other hand, we are careful to preserve at least weak completeness, which means that if search fails then no proof can exist. Thus we can rely on success due to soundness and failure due to weak completeness, while we have no information if the program does not terminate. This summarizes some essential differences between logic programming and general theorem proving.

The idea of logical framework implementations such as λProlog and Elf is to use the operational reading of specifications to implement algorithms for proof search and related problems. In many cases, the original specification itself can be used algorithmically. For example, a natural semantics specification of Mini-ML [Hannan 1991, Michaylov and Pfenning 1991] can be used directly for evaluation or type-checking, one of the original motivations for natural semantics [Kahn 1987, Hannan 1993].

We base our operational understanding of logic programming on the sequent calculus. The operational interpretation of a logical specification is based on two principles: goal-directed search [Miller et al. 1991] and focusing [Andreoli 1992]. Goal-directed search expresses that we always first apply the right rules bottom-up to derive a given sequent until the succedent is atomic. An atomic succedent should now result in an analogue to procedure call. This is achieved by focusing on a particular hypothesis and applying a succession of left rules until it is atomic. If it then happens to unify with the atomic succedent we next attempt to derive the pending premises of the left rules; otherwise we fail and backtrack. In a slight abuse of terminology we refer to derivations which are both goal-directed and focused as *uniform*. If every derivable judgment has a uniform derivation we claim to have an *abstract logic programming language* because search following this operational specification will be sound and weakly complete.

We now specify uniform derivations more concretely, in the form of two mutually recursive judgments for LF.

$$\begin{array}{ll} \Gamma \stackrel{uni}{\Longrightarrow} M : A & A \text{ is uniformly derivable} \\ \Gamma \stackrel{uni}{\Longrightarrow} u{:}A \gg N : P & A \text{ immediately entails } P \end{array}$$

In these judgments, M and N are proof terms for A and P, respectively. In the immediate entailment judgment, A is the hypothesis we have focused on and u its label. When viewed operationally, we think of Γ, A and P as given, while M and N are computed together with the derivation. We presuppose and maintain the

following invariants:

1. $\vdash \Gamma\ Ctx$ in both judgments;
2. $\Gamma \vdash A : \mathsf{type}$ and
3. $\Gamma \vdash M : A$ for uniform derivability, and
4. $\Gamma \vdash P : \mathsf{type}$ and
5. $\Gamma, u{:}A \vdash N : P$ for immediate entailment.

Actually, the restricted form of search guarantees a stronger invariant, namely that M is always canonical and N always atomic.

Atomic judgments.

$$\frac{\Gamma \vdash Q \equiv P : \mathsf{type}}{\Gamma \stackrel{uni}{\Longrightarrow} u{:}Q \gg u : P}\ \mathsf{init}$$

$$\frac{h{:}A \text{ in } \Sigma \text{ or } \Gamma \qquad \Gamma \stackrel{uni}{\Longrightarrow} u{:}A \gg N : P}{\Gamma \stackrel{uni}{\Longrightarrow} [h/u]N : P}\ \mathsf{call}^u$$

Hypothetical judgments.

$$\frac{\Gamma, u{:}A \stackrel{uni}{\Longrightarrow} M : B}{\Gamma \stackrel{uni}{\Longrightarrow} \lambda u{:}A.\, M : A \to B}\ {\to}R^u$$

$$\frac{\Gamma \stackrel{uni}{\Longrightarrow} u{:}B \gg N : C \qquad \Gamma \stackrel{uni}{\Longrightarrow} M : A}{\Gamma \stackrel{uni}{\Longrightarrow} w{:}A \to B \gg [(w\,M)/u]N : C}\ {\to}L^u$$

Parametric judgments.

$$\frac{\Gamma, x{:}A \stackrel{uni}{\Longrightarrow} M : B}{\Gamma \stackrel{uni}{\Longrightarrow} \lambda x{:}A.\, M : \Pi x{:}A.\, B}\ \Pi R^x$$

$$\frac{\Gamma \vdash M \Uparrow A \qquad \Gamma \stackrel{uni}{\Longrightarrow} u{:}[M/x]B \gg N : C}{\Gamma \stackrel{uni}{\Longrightarrow} w{:}\Pi x{:}A.\, B \gg [(w\,M)/u]N : C}\ \Pi L^u$$

Uniform derivations are sound and complete with respect to sequent derivations. In fact, we can prove a stronger theorem that there is a bijection between canonical objects M of a given type A and the objects such that $\stackrel{uni}{\Longrightarrow} M : A$ is derivable [Pfenning 1991*a*, Dyckhoff and Pinto 1994, Pfenning 2001, Pinto and Dyckhoff 1998].

4.2. THEOREM (Properties of LF uniform derivations).

1. *If* $\Gamma \overset{uni}{\Longrightarrow} M : A$ *then* $\Gamma \vdash M \Uparrow A$.
2. *If* $\Gamma \vdash M \Uparrow A$ *then* $\Gamma \overset{uni}{\Longrightarrow} M : A$.

PROOF. The first property is easy to see by induction on the uniform derivation. The second can be proved by induction on the definition of canonical forms, after appropriate generalization for atomic forms (see [Pfenning 2001]). An alternative proof examines the permutability of inference rules in the sequent calculus for LF from Section 4.1. □

We now revisit the remaining non-deterministic choices we examined in the discussion of tactics in Section 4.2.

Conjunctive choice. We always solve the subderivations in the multiple premise rule $\to L$ from left to right. This means that when a hypothesis $u{:}A \to (B \to C)$ is used to derive C, the first subgoal to be solved is B and the second A. If we rewrite the same declaration with the arrows reversed, we obtain $u : (C \leftarrow B) \leftarrow A$ which lends itself to a natural reading as a labelled program clause in logic programming. Using the convention that "$\leftarrow$" is left-associative, we can write this even more concisely as $u : C \leftarrow B \leftarrow A$. It is important to derive the premises of $\to L$ in this order since we do not want to solve subgoals until we know if the target type (C in the example) matches the atomic goal. In Prolog terminology conjunctive choice is called *subgoal selection*.

Disjunctive choice. We employ deep backtracking as indicated in Section 4.2. Since only one inference rule applies to any sequent, disjunctive choices arise only in two circumstances: we have to decide which constant or hypothesis to use for one of the call rules, and unification may allow more than one possibility (see the notes on existential choice below). We first try constants from first to last in the fixed signature Σ, then the parameters and hypotheses from Γ from right to left (the most recently introduced hypothesis is tried first).

Universal choice. Just as before, we simply introduce new parameters or hypothesis labels.

Existential choice. In the ΠL rule we introduce a fresh meta-variable X, record Γ and A and proceed. When we try to complete a branch of the derivation with the init rule, we use unification instead of equality. λProlog employs Huet's unification algorithms to enumerate pre-unifiers, while Elf uses constraint simplification based on patterns [Dowek et al. 1996].

To illustrate uniform derivations we reconsider the example at the end of Section 3.2 with its encoding in LF from Section 3.4. We omit the proof terms for the sake of brevity.

$\cdot \overset{uni}{\Longrightarrow} \Pi A{:}\mathsf{o}.\, \mathsf{nd}(A\, \mathsf{imp\ not\ not}\ A)$
  ΠR^A which leaves
$A{:}\mathsf{o} \overset{uni}{\Longrightarrow} \mathsf{nd}(A\, \mathsf{imp\ not\ not}\ A)$
  call with impi which leaves
$A{:}\mathsf{o} \overset{uni}{\Longrightarrow} (\Pi A{:}\mathsf{o}.\, \Pi B{:}\mathsf{o}.\, (\mathsf{nd}(A) \to \mathsf{nd}(B)) \to \mathsf{nd}(A\, \mathsf{imp}\, B)) \gg \mathsf{nd}(A\, \mathsf{imp\ not\ not}\ A)$
  ΠL with A which leaves
$A{:}\mathsf{o} \overset{uni}{\Longrightarrow} (\Pi B{:}\mathsf{o}.\, (\mathsf{nd}(A) \to \mathsf{nd}(B)) \to \mathsf{nd}(A\, \mathsf{imp}\, B)) \gg \mathsf{nd}(A\, \mathsf{imp\ not\ not}\ A)$
  ΠL with $\mathsf{not\ not}\ A$ which leaves
$A{:}\mathsf{o} \overset{uni}{\Longrightarrow} ((\mathsf{nd}(A) \to \mathsf{nd}(\mathsf{not\ not}\ A)) \to \mathsf{nd}(A\, \mathsf{imp\ not\ not}\ A)) \gg \mathsf{nd}(A\, \mathsf{imp\ not\ not}\ A)$
  $\to L$ which leaves two subgoals
$A{:}\mathsf{o} \overset{uni}{\Longrightarrow} \mathsf{nd}(A\, \mathsf{imp\ not\ not}\ A) \gg \mathsf{nd}(A\, \mathsf{imp\ not\ not}\ A)$
  init which is solved, leaving one subgoal
$A{:}\mathsf{o} \overset{uni}{\Longrightarrow} \mathsf{nd}(A) \to \mathsf{nd}(\mathsf{not\ not}\ A)$

In the remainder we omit the immediate entailment steps.

$A{:}\mathsf{o} \overset{uni}{\Longrightarrow} \mathsf{nd}(A) \to \mathsf{nd}(\mathsf{not\ not}\ A)$	$\to R^u$
$A{:}\mathsf{o}, u{:}\mathsf{nd}(A) \overset{uni}{\Longrightarrow} \mathsf{nd}(\mathsf{not\ not}\ A)$	call with noti
$A{:}\mathsf{o}, u{:}\mathsf{nd}(A) \overset{uni}{\Longrightarrow} \Pi p{:}o.\, (\mathsf{nd}(\mathsf{not}\ A) \to \mathsf{nd}(p))$	ΠR^p
$A{:}\mathsf{o}, u{:}\mathsf{nd}(A), p{:}o \overset{uni}{\Longrightarrow} (\mathsf{nd}(\mathsf{not}\ A) \to \mathsf{nd}(p))$	$\to R^w$
$A{:}\mathsf{o}, u{:}\mathsf{nd}(A), p{:}\mathsf{o}, w{:}\mathsf{nd}(\mathsf{not}\ A) \overset{uni}{\Longrightarrow} \mathsf{nd}(p)$	call with note, leaving subgoals
$A{:}\mathsf{o}, u{:}\mathsf{nd}(A), p{:}\mathsf{o}, w{:}\mathsf{nd}(\mathsf{not}\ A) \overset{uni}{\Longrightarrow} \mathsf{nd}(\mathsf{not}\ A)$	call with w, solved, and
$A{:}\mathsf{o}, u{:}\mathsf{nd}(A), p{:}\mathsf{o}, w{:}\mathsf{nd}(\mathsf{not}\ A) \overset{uni}{\Longrightarrow} \mathsf{nd}(A)$	call with u, solved

To compute the proof term we proceed through the sequents, assigning proof terms at each step. At the root, this yields the sequent

$$A{:}\mathtt{o} \overset{uni}{\Longrightarrow} \mathtt{(impi\ (}\lambda u\mathtt{:nd}\ A\mathtt{.\ noti\ (}\lambda p\mathtt{:o.}\ \lambda w\mathtt{:nd\ (not}\ A\mathtt{).\ note}\ w\ p\ u\mathtt{)))} \\ \mathtt{:\ nd\ (}A\ \mathtt{imp\ not\ not}\ A\mathtt{).}$$

There are some advantages and some disadvantages to the logic programming approach to meta-programming. Perhaps the most important advantage is uniformity of language for specification and implementation. Specific algorithms such a evaluation, type inference, or certain theorem proving strategies can easily be implemented at a very high level. On the other hand, the logic programming paradigm does not lend itself very well to interactive theorem proving since the state of the search and user commands are inherently imperative in nature. In λProlog this is addressed with extra-logical constructs which augment the logical foundation, just as Prolog extends Horn logic in numerous ways. Furthermore, the current state of the art in implementation of λProlog is such that complex tactics or decision procedures can be much faster in a functional meta-language. An ongoing effort in compiler design and implementation might change this situation in the near future [Nadathur and Mitchell 1999].

Elf remains pure and is therefore difficult to use for interactive theorem proving. However the purity of the language has an important benefit, namely that we can express proofs of meta-theorems to a certain extent. In particular, we can write meta-programs in Elf which translate traces of a search algorithm written in Elf to deductions as specified in LF. We will see an example for this kind of application in the Section 5.

4.5. Theory development

In practical applications one is usually interested in more than just proving one theorem, but in the development of a whole theory consisting of declarations, definitions, lemmas, and theorems. Moreover, theories are often organized into subtheories related in a variety of ways.

At the most fundamental level, the logical framework calculus LF can be extended by global definitions of the form $c{:}A = M$ or by local definitions in the form $\mathbf{let}\ x{:}A = M\ \mathbf{in}\ N$. These can be viewed as either introducing syntactic abbreviations (if the type A represents a syntactic category) or introducing a derived rule A with derivation M (if the type A represents a judgment). One can either view such an extension as semantically completely transparent so that the **let** above is treated as syntactic sugar for $(\lambda x{:}A.\,N)\,M$, or one can introduce a new typing rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x{:}A \vdash N : C}{\Gamma \vdash \mathbf{let}\ x{:}A = M\ \mathbf{in}\ N : C}\ \mathsf{let}$$

and a new rule of definitional equality

$$\mathbf{let}\ x{:}A = M\ \mathbf{in}\ N \equiv [M/x]N.$$

The canonical form theorem and decidability of type-checking continue to hold, but the search operations underlying both tactics and logic programming are complicated. The problem is that expansion of all definitions is rarely feasible, while not expanding them jeopardizes weak completeness. A solution of this problem for LF based on a simple form of strictness analysis is proposed in [Pfenning and Schürmann 1998*a*].

In the sequent calculus, the introduction of a lemma into the derivation during search corresponds to an application of the cut rule.

$$\frac{\Gamma \overset{LF}{\Longrightarrow} M : A \qquad \Gamma, u : A \overset{LF}{\Longrightarrow} N : C}{\Gamma \overset{LF}{\Longrightarrow} \mathbf{let}\ u{:}A = M\ \mathbf{in}\ N : C}\ \mathsf{Cut}^u$$

One could also choose the proof term $[M/u]N$ in the conclusion in order to avoid a language extension. The cut rule for LF is *admissible*, which means that any instance of this rule can be eliminated from a derivation.

For further discussion of modularity mechanisms in logical frameworks, see Section 8.1.

5. Representing meta-theory

Logical frameworks are designed to admit a direct and natural representation of deductive systems at a very high level of abstraction. In Section 3 we showed that checking the validity of a derivation can be reduced to type-checking in the framework which is decidable. In Section 4 we indicated how generic ideas for proof search in a logical framework can support theorem proving in particular logics, and how a logic programming interpretation of a framework can be used for the implementation of specific algorithms related to deductive systems.

This leaves the question if we can take advantage of the conciseness and elegance of the encodings to also mechanize the meta-theory of deductive systems. For example, we might want to prove that the natural deduction formulation of intuitionistic logic in Section 3.2 and the axiomatic formulation in Section 3.5 have the same theorems. Other examples from the area of logic include admissibility of inference rules such as cut in a sequent system, or the correctness of logical interpretations. In the area of programming languages we think of properties such as type preservation, correctness of type inference algorithms, or compiler correctness.

The answer is a qualified "*yes*". Some frameworks such as FS_0 are specifically designed for meta-theoretic reasoning, but they give up techniques such as static proof checking, higher-order abstract syntax, or hypothetical judgments as functions. As we explain below, there are some difficulties with encodings utilizing higher-order abstract syntax with a number of possible solutions. In many ways the potential of logical frameworks for meta-theoretic reasoning has not yet been fully explored.

Just as we isolated the notions of variable binding, parametric, and hypothetical judgments as central in the presentation of deductive systems, we should analyze the proof techniques used to carry out the meta-theory of deductive systems and then consider how a framework might support them. By far the most common proof technique is induction, both over the structure of expressions and derivations. Thus one naturally looks towards frameworks that permit inductive definitions of judgments and allow the corresponding induction principles. Unfortunately, there is a conflict between induction and the representation techniques of higher-order abstract syntax and functional representation of hypothetical judgments. The issue is complicated further by dependent types, so we consider first the implicational fragment of the simply-typed representation of deductions.

$$\begin{array}{lcl} \mathsf{nd} & : & \mathsf{type} \\ \mathsf{impi} & : & (\mathsf{nd} \to \mathsf{nd}) \to \mathsf{nd} \\ \mathsf{impe} & : & \mathsf{nd} \to \mathsf{nd} \to \mathsf{nd} \end{array}$$

Even if we considered the above signature as complete (rather than open-ended), the type nd would not be inductively defined in the usual sense, because of the negative occurrence of nd in the type of impi. Straightforward attempts to formulate a valid induction principle for the type nd fail. Informally, at least one difficulty is clear: when we try to prove a theorem about natural deductions, we invariably have to generalize over all possible collection of hypotheses. Since they are not represented explicitly in our technique, we cannot directly formulate the required induction proofs. We consider an example below.

There is a further difficulty with induction in the framework which stems from the essential open-endedness of representations. For example, assume we declare constants z for zero and s for successor in the formulation of first-order logic, but we do not assume an induction principle for natural numbers in our object logic. If the framework permitted an induction principle over the representation type i, we would no longer have an adequate encoding of first-order logic with two uninterpreted function constants. The encoding of the universal introduction rule,

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

now represents an ω-rule, since objects of type $\Pi a{:}\mathsf{i}.\, \mathsf{nd}\ (A\ a)$ allow case analysis on a and are therefore no longer necessarily parametric in a. Depending on the strength of the induction principle in the meta-language we would be able to derive various propositions in the object language that are not actually derivable in pure first-order logic and the adequacy of the representation is destroyed. A similar problem already arises at the level of syntax if we permit primitive recursion into the logical framework.

Several options have been explored to escape this dilemma. The first is to reject the notion of higher-order abstract syntax and use inductive representations directly (see, for example, [Matthews et al. 1993, Basin and Constable 1993, Feferman 1988, Magnusson and Nordström 1994]). This engenders a complication of the encoding and consequently of the meta-theory, which now has to deal with many lemmas regarding variable naming. This can be alleviated by using de Bruijn indices [de Bruijn 1972], yet formalizations are still substantially more complex than informal proofs. There are many examples of formal developments along these lines.

A second possibility is to relax the conditions on inductive definitions, which leads to *partial inductive definitions* [Hallnäs 1991]. They allow inversion principles but not a direct generalization of proofs by induction. Partial inductive definitions have been used as the basis for a logical framework [Hallnäs 1987, Eriksson 1993*a*], implemented in the Pi derivation editor [Eriksson 1994]. Their potential for formalizing meta-theory is currently being explored by McDowell and Miller [1997] (see also [McDowell 1997]); more on their approach below.

A third option is to employ reflection with some restrictions to ensure soundness. In [Despeyroux, Pfenning and Schürmann 1997] this was achieved by a modal type operator satisfying the laws of S4. However, the practicality of these and some related proposals [Despeyroux and Hirschowitz 1994, Despeyroux, Felty and Hirschowitz 1995, Leleu 1998] has never been demonstrated. Different reflection mechanisms have been employed in the Calculus of Construction [Rueß 1996, Rueß 1997] and Nuprl [Allen, Constable, Howe and Aitken 1990]. These last two do not use higher-order abstract syntax.

A fourth option is to externalize the induction. This leads to a three-level architecture: the object logic, the logical framework in which it is specified, and a meta-logic for reasoning about the logical framework. Variations of this are currently pursued by McDowell and Miller [1997] and Schürmann and Pfenning [1995, 1998]. In principle, any meta-logic could be used for reasoning about the logical framework, but the effort required to develop the theory of the framework and then apply it to individual signatures would be prohibitive unless the meta-logic was specifically designed for meta-theoretic reasoning. Briefly, the logic of McDowell and Miller is based on definitional reflection [Schroeder-Heister 1993] and natural number induction, while that of Schürmann and Pfenning admits only $\forall\exists$ formulas where the quantifiers range over closed LF objects and uses explicit termination orderings [Rohwedder and Pfenning 1996]. Recently, this approach has been generalized by Schürmann [2000].

A more detailed discussion of such meta-logical frameworks is beyond the scope of this chapter. In the next section we present another approach where the meta-theory is only partially verified, but where the computational contents of the meta-theoretic proofs is directly available for execution.

5.1. Relational meta-theory

As alluded to above, it is difficult to soundly extend the logical framework to include induction. However, it is possible to encode the computational contents of proofs of meta-theoretic properties in Elf and thereby partially verify them. Moreover, they can be executed for a number of different purposes. The technique employs higher-level judgments as introduced in Section 3.6.

As an example we consider the equivalence between natural deduction and axiomatic formulations of the fragment of first-order logic introduced in Sections 3.2 and 3.5. In one direction this is expressed simply as:

If $\vdash^{\!A} A$ *then* $\vdash^{\!N} A$.

Recall that formulas A are represented as objects of type o, while derivations of $\vdash^{\!A} A$ are represented by objects of type $\mathsf{hil}\ulcorner A \urcorner$ and derivations of $\vdash^{\!N} A$ as objects of type $\mathsf{nd}\ulcorner A \urcorner$. Expressed in a meta-logic for LF, we can use adequacy of the encodings to reformulate the theorem.

For any LF objects $A : \mathsf{o}$ *and* $H : \mathsf{hil}\, A$ *there exists an LF object* $D : \mathsf{nd}\, A$.

If we ignore the issues of parameters for the moment, the quantifiers range over closed objects with respect to the signature that encodes natural and axiomatic formulations of intuitionistic logic. From a constructive proof of this proposition we can extract a function which maps a formula A and a derivation of $\vdash^{\!A} A$ to a deduction of $\vdash^{\!N} A$. If this function were representable in the logical framework, it would have type

$$\Pi A{:}\mathsf{o}.\, \mathsf{hil}\, A \to \mathsf{nd}\, A.$$

Since the proof proceeds by induction over the structure of the axiomatic derivation $\mathcal{H}$ of $\vdash^{A} A$, such a function would be defined by induction over its second argument—something the framework does not allow. However, we can specify this function as a higher-level judgment relating $\mathcal{H}$ and the natural deduction $\mathcal{D}$. This higher-level judgment is declared as a type family hilnd.

$$\mathsf{hilnd} : \Pi A{:}\mathsf{o}.\, \mathsf{hil}\, A \to \mathsf{nd}\, A \to \mathsf{type}$$

This relation can be specified in LF and executed as a logic program in Elf. Queries have the form $\mathsf{hilnd}\, A\, H\, \mathbf{D}$, where A and H are given closed objects of appropriate type, while $\mathbf{D}$ is a free variable which will be computed during logic programming search.

It is important to realize, however, that type-checking the signature declaring hilnd does not guarantee the validity of the meta-theorem we were trying to prove. For this, some additional conditions have to be satisfied: *mode correctness* which expresses that the logic programming interpretation of hilnd respects the desired input/output interpretation, *termination* which guarantees that each call of hilnd of the form above terminates, and *coverage* which guarantees that for each possible combination of input values a case in the definition of hilnd will be applicable. Some aspects of this check are discussed in [Pfenning and Rohwedder 1992, Rohwedder and Pfenning 1996].

A similar idea in the area of functional programming without the notion of higher-order abstract syntax has been explored in the ALF system [Magnusson 1995, Magnusson and Nordström 1994, Coquand and Smith 1993, Coquand, Nordström, Smith and von Sydow 1994] and the Foetus system [Abel 1999]. The empirical evidence suggests that this shortens developments considerably and allows the formulations of functions in a manner which is closer to functional programming practice [Coquand 1992, Gaspes and Smith 1992, Magnusson 1993]. In these systems, termination and coverage has also been externalized, rather than forcing adherence to an inflexible schema of primitive recursion.

5.2. Translating axiomatic derivations to natural deductions

In this section we illustrate the relational representation of proofs by relating derivations in the axiomatic system to natural deductions. As a first step we prove that every axiomatic deduction may be transformed into a natural deduction.

5.1. THEOREM. *If* $\vdash^{A} A$ *then* $\vdash^{N} A$.

PROOF. The proof proceeds by a simple structural induction over the derivation $\mathcal{H} :: \ \vdash^{A} A$. In each case we exhibit the corresponding natural deduction. Our representation of this proof introduces a new judgment relating, for any formula A, the Hilbert derivations of A to the natural deductions of A. This judgment is represented by the type family

$$\mathsf{hilnd} \quad : \quad \mathsf{hil}\, A \to \mathsf{nd}\, A \to \mathsf{type}$$

where we have left a quantifier over A implicit as explained in Section 3.4. As explained in the preceding section, this relation implements a total function $\Pi A{:}\mathsf{o}.\, \mathsf{hil}\, A \to \mathsf{nd}\, A$ which is not directly expressible in the framework.

Each case in the induction argument turns into a declaration of a corresponding higher-level judgment.

Case:

$$\mathcal{H} = \dfrac{}{\vdash^{A} A \supset (B \supset A)}\, K$$

In this case we have to supply a natural deduction of $\vdash^{N} A \supset (B \supset A)$, which we have already seen at the end of Section 3.4. Recall that k implements the axiom K.

```
hnd_k : hilnd k (impi (λu:nd A. impi (λv:nd B. u))).
```

Case:

$$\mathcal{H} = \dfrac{}{\vdash^{A} (A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))}\, S$$

A natural deduction of the conclusion is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash^{N} A \supset (B \supset C)}\,u \quad \dfrac{}{\vdash^{N} A}\,w}{\vdash^{N} B \supset C}\,{\supset}\mathrm{E} \qquad \dfrac{\dfrac{}{\vdash^{N} A \supset B}\,v \quad \dfrac{}{\vdash^{N} A}\,w}{\vdash^{N} B}\,{\supset}\mathrm{E}}{\vdash^{N} C}\,{\supset}\mathrm{E}}{\vdash^{N} A \supset C}\,{\supset}\mathrm{I}^{w}}{\vdash^{N} (A \supset B) \supset (A \supset C)}\,{\supset}\mathrm{I}^{v}}{\vdash^{N} (A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))}\,{\supset}\mathrm{I}^{u}$$

This deduction can now be represented in LF by the usual method.

```
hnd_s :
   hilnd s
      (impi (λu:nd (A imp B imp C).
               impi (λv:nd (A imp B).
                        impi (λw:nd A. impe (impe u w) (impe v w))))).
```

Case:

$$\mathcal{H} = \dfrac{}{\vdash^{A} (A \supset \neg B) \supset ((A \supset B) \supset (\neg A))}\, N_1$$

This is similar to the previous case.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash^N A \supset \neg B}u \quad \cfrac{}{\vdash^N A}w}{\vdash^N \neg B}\supset\!E \qquad \cfrac{\cfrac{}{\vdash^N A \supset B}v \quad \cfrac{}{\vdash^N A}w}{\vdash^N B}\supset\!E}{\vdash^N p}\neg E}{\vdash^N \neg A}\neg I^{p,w}}{\vdash^N (A \supset B) \supset \neg A}\supset\! I^v}{\vdash^N (A \supset \neg B) \supset ((A \supset B) \supset \neg A)}\supset\! I^u$$

In the formalization, the propositional parameter p appears as a bound variable.

```
hnd_n1 :
   hilnd n1
      (impi (λu:nd (A imp not B).
               impi (λv:nd (A imp B).
         noti (λp:o. λw:nd A. note (impe u w) p (impe v w))))).
```

The remaining axioms are easy to prove, and we only show their encodings

```
hnd_n2 :
   hilnd n2 (impi (λu:nd (not A). impi (λv:nd A. note u B v))).
hnd_f1 :
   hilnd (f1 T) (impi (λu:nd (forall (λx:i. A x)). foralle u T)).
hnd_f2 :
   hilnd f2
      (impi (λu:nd (forall (λx:i. B imp A x)).
               impi (λv:nd B. foralli (λa:i. impe (foralle u a) v)))).
```

Case:

$$\mathcal{H} = \cfrac{\begin{matrix}\mathcal{H}_1 \\ \vdash^A A \supset B\end{matrix} \qquad \begin{matrix}\mathcal{H}_2 \\ \vdash^A A\end{matrix}}{\vdash^A B} MP$$

By induction hypothesis on $\mathcal{H}_1$ and $\mathcal{H}_2$ there exist natural deductions $\mathcal{D}_1 :: \vdash^N A \supset B$ and $\mathcal{D}_2 :: \vdash^N A$, respectively. Using the rule of implication elimination $\supset$E, we obtain

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \vdash^N A \supset B\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \vdash^N A\end{matrix}}{\vdash^N B} \supset\! E$$

In the representation we emphasize the operational reading of the implementation by using the arrow that points to the left. It associates to the left, and therefore $A_3 \leftarrow A_2 \leftarrow A_1$ is equivalent to $A_1 \to A_2 \to A_3$.


```
hnd_mp : hilnd (mp H1 H2) (impe D1 D2)
            ← hilnd H1 D1
            ← hilnd H2 D2.
```

Note that hilnd H_1 D_1 will be the first subgoal to be solved, and hilnd H_2 D_2 the second, according to the operational semantics sketched in Section 4.4.

Case:

$$\mathcal{H} = \cfrac{\begin{matrix}\mathcal{H}_1 \\ \vdash^A [a/x]A\end{matrix}}{\vdash^A \forall x.\, A} UG^a$$

This case corresponds directly to universal introduction ($\forall$I) in natural deduction. By induction hypothesis on $\mathcal{H}_1$ there exists a natural deduction $\mathcal{D}_1 :: \vdash^N [a/x]A$. Since the deduction $\mathcal{D}_1$ is not hypothetical, the side condition on UG that a not appear in A is sufficient to guarantee the corresponding side condition on $\forall$I and we can form

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \vdash^N [a/x]A\end{matrix}}{\vdash^N \forall x.\, A} \forall I^a$$

In the representation, $\mathcal{H}_1$ is a function from a to a deduction of $[a/x]A$. Thus the higher-level judgment relating $\mathcal{H}_1$ to $\mathcal{D}_1$ is parametric in a. Parametric judgments are represented by functions as usual, so a dependent function type will appear in the premise.

```
hnd_ug : hilnd (ug H1) (foralli D1) ← (Πa:i. hilnd (H1 a) (D1 a)).
```

Operationally in Elf, solving the subgoal introduces a new parameter a and substitutes it for the variable bound in H_1. The resulting deduction is translated to a natural deduction that may contain a. Matching this against the pattern $(D_1\, a)$ creates the correct functional representation of the judgment that is hypothetical in a, and which is the premise of $\forall$I and thus the argument to foralli. □

The proof above describes a method for translating axiomatic derivations to natural deductions. Under the Curry-Howard isomorphism [Howard 1980], this corresponds to a translation from typed combinators (based on S and K and others) to typed λ-terms. As a sample execution of this program, consider the query

```
hilnd (mp (mp s k) k) D
```

where $\mathbf{D}$ is a free variable of type nd $(A\,\mathsf{imp}\,A)$. This will compute the following instantiation for $\mathbf{D}$, which is an indirect way of deriving $\vdash^N A \supset A$.

```
impe
  (impe
     (impi
        (λu:nd (A imp (B imp A) imp A).
           impi
              (λv:nd (A imp B imp A).
                 impi (λw:nd A. impe (impe u w)
                                     (impe v w)))))
     (impi (λu:nd A. impi (λv:nd (B imp A). u))))
  (impi (λu:nd A. impi (λv:nd B. u))).
```

5.3. *The deduction theorem*

One crucial step in proving the other direction (natural deductions can be translated to axiomatic derivations) is the *deduction theorem*. In its simplest form it concerns a hypothetical derivation: if we can prove B assuming A (written as $A \vdash^{A} B$), then we can derive $\vdash^{A} A \supset B$. This is not quite enough for our application, since during a natural deduction many hypotheses may arise. So we let Δ range over collections of hypotheses $A_1, \ldots, A_n$ and write $\Delta \vdash^{A} B$. An implementation of a proof of the deduction theorem using FS_0 is described in [Basin and Matthews 1996] and may be compared to the relational implementation below.

5.2. THEOREM (Deduction Theorem). *If* $\Delta, A \vdash^{A} B$ *then* $\Delta \vdash^{A} A \supset B$.

PROOF. The proof proceeds by induction on the structure of the derivation $\mathcal{H} :: \Delta, A \vdash^{A} B$. In the implementation of the proof the extraneous hypotheses Δ will be represented by hypotheses in LF and can therefore be left implicit in the main judgment. Thus the proof is implemented as a higher-level judgment, relating the representation of the hypothetical derivation of $A \vdash^{A} B$ to the derivation of $\vdash^{A} A \supset B$. Recall that a hypothetical derivation is represented as an LF function from derivations of the hypothesis to derivations of the conclusion. Thus we arrive at the type family

```
ded : (hil A → hil B) → hil (A imp B) → type
```

where A and B are implicitly quantified.

Case: $\mathcal{H} = \Delta, A \vdash^{A} A$, that is, $\mathcal{H}$ consists of a use of the hypothesis A. Then we need to show that $\Delta \vdash^{A} A \supset A$. This follows by two applications of Modus Ponens from (S) and (K). Written in linear form instead of the more awkward tree we have

$$\begin{array}{lll}
1 & (A \supset ((B \supset A) \supset A)) \supset ((A \supset (B \supset A)) \supset (A \supset A)) & S \\
2 & (A \supset ((B \supset A) \supset A)) & K \\
3 & (A \supset (B \supset A)) \supset (A \supset A) & MP\,1\,2 \\
4 & A \supset (B \supset A) & K \\
5 & A \supset A & MP\,3\,4
\end{array}$$

As an LF term, this is represented succinctly by `mp (mp s k) k`, a term already familiar from the sample query at the end of the previous section. The LF function λu`:hil` A`.` u represents the immediate use of the hypothesis $\vdash^{A} A$, labelled internally by u. Thus we have

```
ded_id : ded (λu:hil A. u) (mp (mp s k) k).
```

Case: $\mathcal{H} = \Delta, A \vdash^{A} A_i$, where A_i occurs in Δ. In this case we have to give a derivation of $\Delta \vdash^{A} A \supset A_i$. But this follows from an application of Modus Ponens and K.

$$\begin{array}{lll}
1 & \Delta \vdash^{A} A_i \supset (A \supset A_i) & K \\
2 & \Delta \vdash^{A} A_i & (hyp) \\
3 & \Delta \vdash^{A} A \supset A_i & MP\,1\,2
\end{array}$$

There is no corresponding case in the implementation of the type family `ded`. Instead, we need to make the assumption that the deduction theorem applied to a new hypothesis labelled w yields `mp k` w wherever w is introduced. This technique will be illustrated in the next section.

Case:

$$\mathcal{H} = \frac{}{\Delta, A \vdash^{A} B_1 \supset (B_2 \supset B_1)}\,K$$

Then we proceed as follows:

$$\begin{array}{lll}
1 & \Delta \vdash^{A} (B_1 \supset (B_2 \supset B_1)) \supset (A \supset (B_1 \supset (B_2 \supset B_1))) & K \\
2 & \Delta \vdash^{A} B_1 \supset (B_2 \supset B_1) & K \\
3 & \Delta \vdash^{A} A \supset (B_1 \supset (B_2 \supset B_1)) & MP\,1\,2
\end{array}$$

```
ded_k : ded (λu:hil A. k) (mp k k).
```

Cases: All remaining axioms (S, N_1, N_2, F_1, F_2) are handled as in the previous case. We only show their implementations.

```
ded_n1 : ded (λu:hil A. n1) (mp k n1).
ded_n2 : ded (λu:hil A. n2) (mp k n2).
ded_f1 : ded (λu:hil A. f1 T) (mp k (f1 T)).
ded_f2 : ded (λu:hil A. f2) (mp k f2).
```

Case:

$$\mathcal{H} = \frac{\begin{array}{c}\mathcal{H}_1 \\ \Delta, A \vdash^{A} B_1 \supset B_2\end{array} \qquad \begin{array}{c}\mathcal{H}_2 \\ \Delta, A \vdash^{A} B_1\end{array}}{\Delta, A \vdash^{A} B_2}\,MP$$

1	$\Delta \vdash^{A} A \supset (B_1 \supset B_2)$	Ind. hyp. on $\mathcal{H}_1$
2	$\Delta \vdash^{A} (A \supset (B_1 \supset B_2)) \supset ((A \supset B_1) \supset (A \supset B_2))$	S
3	$\Delta \vdash^{A} (A \supset B_1) \supset (A \supset B_2)$	$MP\ 2\ 1$
4	$\Delta \vdash^{A} A \supset B_1$	Ind. hyp. on $\mathcal{H}_2$
5	$\Delta \vdash^{A} A \supset B_2$	$MP\ 3\ 4$

Appeals to induction hypotheses are implemented in the premises of the higher level judgment, generating H_1' and H_2', respectively. Note how the premises $\mathcal{H}_1$ and $\mathcal{H}_2$ of $\mathcal{H}$ are once again hypothetical, that is, they may depend on the assumption A. This is implemented as $(H_1\ u)$ and $(H_2\ u)$ in the declaration below.

```
ded_mp :
    ded (λu:hil A. mp (H1 u) (H2 u)) (mp (mp s H1') H2')
       ← ded H1 H1'
       ← ded H2 H2'.
```

Case:

$$\mathcal{H} = \dfrac{\begin{array}{c}\mathcal{H}_1\\ \Delta, A \vdash^{A} [a/x]B_1\end{array}}{\Delta, A \vdash^{A} \forall x.\, B_1}\, UG^a$$

1	$\Delta \vdash^{A} A \supset [a/x]B_1$	Ind. hyp. on $\mathcal{H}_1$
2	$\Delta \vdash^{A} \forall x.\,(A \supset B_1)$	$UG^a\ 1$
3	$\Delta \vdash^{A} (\forall x.\,(A \supset B_1)) \supset (A \supset \forall x.\, B_1)$	F_2
4	$\Delta \vdash^{A} A \supset \forall x.\, B_1$	$MP\ 3\ 2$

The side conditions on UG^a and F_2 are satisfied by virtue of the proviso that a not occur in Δ, A, or $\forall x.\, B_1$, that is, that $\mathcal{H}_1$ be parametric in a. In the implementation we simply create a new parameter a.

```
ded_ug :
    ded (λu:hil A. ug (H1 u)) (mp f2 (ug H1'))
       ← (Πa:i. ded (λu:hil A. H1 u a) (H1' a)).
```

□

The declarations for the higher-level judgment ded can be executed as a logic program, thus capturing the computational contents of the deduction theorem. This corresponds to the algorithm for *bracket abstraction* in combinatory logic [Curry and Feys 1958].

5.4. Translating natural deductions to axiomatic derivations

Obtaining a translation from natural deductions to axiomatic derivations is now straightforward. Note that we must allow for hypotheses, since the $\supset$I rule intro-

duces them (if viewed from the bottom up).

5.3. THEOREM. *If $\vdash^{N} A$ follows from hypotheses $\vdash^{N} A_1, \ldots, \vdash^{N} A_n$, then there exists a hypothetical axiomatic derivation of $A_1, \ldots, A_n \vdash^{A} A$.*

PROOF. By induction on $\mathcal{D} :: \vdash^{N} A$. We abbreviate $A_1, \ldots, A_n$ by Δ. In the implementation we deal with each hypothesis as it is introduced, rather than globally. Thus the type family that implements the meta-proof just relates a natural deduction to a Hilbert derivation.

```
ndhil : ΠA:o. nd A → hil A → type.
```

Case:

$$\mathcal{D} = \dfrac{}{\vdash^{N} A_i}\, u_i$$

This constitutes application of a hypothesis. Then $\mathcal{H}$ is a one-step derivation using the corresponding the hypothesis. It is implemented wherever hypotheses are introduced, which are the cases for $\supset$I and $\neg$I.

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\dfrac{}{\vdash^{N} A_1}\, u\\ \mathcal{D}_1\\ \vdash^{N} A_2\end{array}}{\vdash^{N} A_1 \supset A_2}\, \supset\mathrm{I}^u$$

By induction hypothesis on $\mathcal{D}_1$, there exists a derivation $\mathcal{H}_1$ of $\Delta, A_1 \vdash^{A} A_2$. Hence, by the deduction theorem, there exists a derivation $\mathcal{H}_1'$ of $\Delta \vdash^{A} A_1 \supset A_2$, which is what we needed to show. The implementation combines this and the previous case by introducing hypotheses $u{:}\mathsf{nd}\,A_1$ and $v{:}\mathsf{hil}\,A_1$ and assuming that the translation of u should be v. Since this rule introduces a new hypothesis $\vdash^{A} A_1$, we must also indicate how the deduction theorem behaves on the new assumption. This may be gleaned from the second case in the proof of the deduction theorem.

```
ndh_impi :
    ndhil (impi D1) H1'
       ← (Πu:nd A1. Πv:hil A1.
             (ΠC:o. ded (λw:hil C. v) (mp k v))
               → ndhil u v
               → ndhil (D1 u) (H1 v))
       ← ded H1 H1'.
```

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \vdash^{N} \forall x.\, A_1\end{array}}{\vdash^{N} [t/x]A_1}\,\forall\mathrm{E}$$

By induction hypothesis on $\mathcal{D}_1$ there exists a derivation $\mathcal{H}_1$ of $\Delta \vdash^{A} \forall x.\, A_1$. By modus ponens from an instance of axiom schema F_1 and $\mathcal{H}_1$ we can then construct a derivation $\mathcal{H}$ of $\Delta \vdash^{A} [t/x]A_1$.

$$\mathsf{ndh\_foralle}\ :\ \mathsf{ndhil}\ (\mathsf{foralle}\ D_1\ \ T)\ \ (\mathsf{mp}\ \ (\mathsf{f}_1\ \ T)\ \ H_1)\ \leftarrow\ \mathsf{ndhil}\ D_1\ \ H_1.$$

Cases: We omit the remaining cases which are similar to the two given above. It is an instructive exercise to reconstruct the informal argument from the implementation given below.

$$\begin{array}{l}
\mathsf{ndh_impe}\ :\ \mathsf{ndhil}\ (\mathsf{impe}\ D_1\ \ D_2)\ \ (\mathsf{mp}\ \ H_1\ \ H_2)\\
\qquad\qquad \leftarrow \mathsf{ndhil}\ D_1\ \ H_1\\
\qquad\qquad \leftarrow \mathsf{ndhil}\ D_2\ \ H_2.\\
\mathsf{ndh_noti}\ :\\
\quad \mathsf{ndhil}\ (\mathsf{noti}\ D_1)\ \ (\mathsf{mp}\ \ (\mathsf{mp}\ \ \mathsf{n}_1\ \ H_1{}')\ \ H_1{}'')\\
\qquad \leftarrow (\Pi p{:}\mathsf{o}.\ \Pi u{:}\mathsf{nd}\ A_1.\ \Pi v{:}\mathsf{hil}\ A_1.\\
\qquad\qquad (\Pi C{:}\mathsf{o}.\ \mathsf{ded}\ (\lambda w{:}\mathsf{hil}\ C.\ v)\ \ (\mathsf{mp}\ \ \mathsf{k}\ \ v))\\
\qquad\qquad\quad \rightarrow \mathsf{ndhil}\ u\ v\\
\qquad\qquad\quad \rightarrow \mathsf{ndhil}\ (D_1\ \ p\ \ u)\ \ (H_1\ \ p\ \ v))\\
\qquad \leftarrow \mathsf{ded}\ (H_1\ \ (\mathsf{not}\ \ A))\ \ H_1{}'\\
\qquad \leftarrow \mathsf{ded}\ (H_1\ \ A)\ \ H_1{}''.\\
\mathsf{ndh_note}\ :\ \mathsf{ndhil}\ (\mathsf{note}\ D_1\ \ C\ \ D_2)\ \ (\mathsf{mp}\ \ (\mathsf{mp}\ \ \mathsf{n}_2\ \ H_1)\ \ H_2)\\
\qquad\qquad \leftarrow \mathsf{ndhil}\ D_1\ \ H_1\\
\qquad\qquad \leftarrow \mathsf{ndhil}\ D_2\ \ H_2.\\
\mathsf{ndh_foralli}\ :\ \mathsf{ndhil}\ (\mathsf{foralli}\ D_1)\ \ (\mathsf{ug}\ \ H_1)\\
\qquad\qquad \leftarrow (\Pi a{:}\mathsf{i}.\ \mathsf{ndhil}\ (D_1\ \ a)\ \ (H_1\ \ a)).
\end{array}$$

□

In summary, we can represent some aspects of constructive meta-theoretic proofs as higher-level judgments in LF. These higher-level judgments can be executed in Elf with the operational semantics from Section 4.4 to translate derivations between deductive systems. While the result of each individual computation of this form is guaranteed to be correct, the higher-level judgment is only partially verified since termination and coverage of all possible cases are properties outside the scope of the type-checker.

6. Appendix: the simply-typed λ-calculus

For the representation of the abstract syntax of a language, the simply-typed λ-calculus ($\lambda^{\rightarrow}$) is usually adequate. When we tackle the task of representing inference

rules, we will have to refine the type system by adding dependent types. The reader should bear in mind that $\lambda^{\rightarrow}$ should *not* be considered as a functional programming language, but as a representation language. In particular, the absence of recursion will be crucial in order to guarantee adequacy of representations. Our formulation of the simply-typed λ-calculus has two levels: the level of *types* and the level of *objects*, where types classify objects. Furthermore, we have *signatures* which declare type and object constants, and *contexts* which assign types to variables. The presentation is in the style of Church: Every valid object has a unique type. This requires that types appear in the syntax of objects to resolve the inherent ambiguity of certain functions such as the identity function. We let a range over type constants, c over object constants, x over variables.

$$\begin{array}{llcl}
\text{Types} & A & ::= & a \mid A_1 \rightarrow A_2\\
\text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\, M \mid M_1\ M_2\\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}\mathsf{type} \mid \Sigma, c{:}A\\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}$$

We make the general restriction that constants and variables can occur at most once in a signature or context, respectively. We use A and B to range over types, and M and N to range over objects. We refer to type constants a as *atomic types* and types of the form $A \rightarrow B$ as *function types*. We also consider terms that differ only in the names of their bound variables as identical and use the variable convention as for first-order logic in Section 2.

The judgments defining $\lambda^{\rightarrow}$ are

$$\begin{array}{ll}
\vdash_\Sigma A : \mathsf{type} & A \text{ is a valid type}\\
\Gamma \vdash_\Sigma M : A & M \text{ is a valid object of type } A \text{ in context } \Gamma\\
\vdash_\Sigma \Gamma\ \mathit{Ctx} & \Gamma \text{ is a valid context}\\
\vdash \Sigma\ \mathit{Sig} & \Sigma \text{ is a valid signature}
\end{array}$$

Note that the first three of these judgments depend on a signature Σ which we presuppose to be valid. Similarly, we assume that Γ is always valid in the judgment $\Gamma \vdash_\Sigma M : A$. The judgments are defined via the following inference rules.

Valid objects

$$\dfrac{c{:}A \text{ in } \Sigma}{\Gamma \vdash_\Sigma c : A}\,\mathsf{con} \qquad \dfrac{x{:}A \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : A}\,\mathsf{var}$$

$$\dfrac{\vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\, M : A \rightarrow B}\,\mathsf{lam}$$

$$\dfrac{\Gamma \vdash_\Sigma M : A \rightarrow B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : B}\,\mathsf{app}$$

Valid types

$$\frac{a{:}\mathsf{type}\ \text{in}\ \Sigma}{\vdash_\Sigma a : \mathsf{type}}\,\mathsf{con} \qquad \frac{\vdash_\Sigma A : \mathsf{type} \qquad \vdash_\Sigma B : \mathsf{type}}{\vdash_\Sigma A \to B : \mathsf{type}}\,\mathsf{arrow}$$

Valid signatures

$$\frac{}{\vdash \cdot\ \mathit{Sig}}\,\mathsf{sigemp} \qquad \frac{\vdash \Sigma\ \mathit{Sig}}{\vdash \Sigma, a{:}\mathsf{type}\ \mathit{Sig}}\,\mathsf{sigtyp}$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \qquad \vdash_\Sigma A : \mathsf{type}}{\vdash \Sigma, c{:}A\ \mathit{Sig}}\,\mathsf{sigobj}$$

Valid contexts

$$\frac{}{\vdash_\Sigma \cdot\ \mathit{Ctx}}\,\mathsf{ctxemp} \qquad \frac{\vdash_\Sigma \Gamma\ \mathit{Ctx} \qquad \vdash_\Sigma A : \mathsf{type}}{\vdash_\Sigma \Gamma, x{:}A\ \mathit{Ctx}}\,\mathsf{ctxobj}$$

The rules for valid objects are somewhat non-standard in that they contain no check whether the signature Σ or the context Γ are valid, which we presuppose. Furthermore, the rules guarantee that if we have a derivation $\mathcal{D}$ of $\Gamma \vdash_\Sigma M : A$ and Γ is valid, then every context appearing in $\mathcal{D}$ is also valid. This is because the type A in the lam rule is checked for validity as it is added to the context.

Our formulation of the simply-typed λ-calculus above is parametrized by a signature in which new constants can be declared; only variables, λ-abstraction, and application are built into the language itself. The analogue of *observable values* in functional programming languages is the notion of *canonical form*, since they are in one-one correspondence with the data we are trying to represent. Unlike in functional languages, every well-typed object will have an equivalent canonical form which can be calculated with a simple algorithm. For the definition of canonical forms as a deductive system we need two mutually recursive judgments: canonical and atomic forms. For the sake of brevity, we elide the fixed signature Σ from this judgment.

$\Gamma \vdash M \Uparrow A$	object M is canonical of type A
$\Gamma \vdash M \downarrow A$	object M is atomic of type A

An atomic form is a variable or constant applied to some number of arguments, each of which is in canonical form. A canonical form of functional type must be a λ-abstraction; a canonical form of atomic type a must itself be atomic. This is captured with the following inference rules.

$$\frac{\Gamma, x{:}A \vdash M \Uparrow B}{\Gamma \vdash \lambda x{:}A.\, M \Uparrow A \to B}\,\mathsf{arrow} \qquad \frac{\Gamma \vdash M \downarrow a}{\Gamma \vdash M \Uparrow a}\,\mathsf{coerce}$$

$$\frac{x{:}A\ \text{in}\ \Gamma}{\Gamma \vdash x \downarrow A}\,\mathsf{var} \qquad \frac{c{:}A\ \text{in}\ \Sigma}{\Gamma \vdash c \downarrow A}\,\mathsf{con}$$

$$\frac{\Gamma \vdash M \downarrow B \to A \qquad \Gamma \vdash N \Uparrow B}{\Gamma \vdash M\,N \downarrow A}\,\mathsf{app}$$

The algorithm for conversion to canonical and atomic forms introduces λ-abstractions if the object is of functional type, essentially applying η-expansion. At base type we check if the object has the form of a variable or constant applied to some arguments. If so, we convert the arguments to canonical form. If not, we repeatedly apply weak head reduction until the other case applies. This method of definition of a typed λ-calculus corresponds to an operational semantics for a functional language and is very much in the spirit of the method of algorithmic definition for type theories [de Bruijn 1993]. Related systems have been described in [Felty and Miller 1990, Coquand 1991]. The algorithm is given as a deductive system consisting of three judgments which may be interpreted as a logic program.

$M \xrightarrow{whr} M'$	M weak head reduces to M'
$\Gamma \vdash M \Uparrow M' : A$	M converts to canonical form M' at type A
$\Gamma \vdash M \downarrow M' : A$	M converts to atomic form M' at type A

First, the rules for weak head reduction. We write $[N/x]M$ for the result of substituting N for x in M, possibly renaming bound variables to avoid variable capture.

$$\frac{}{(\lambda x{:}A.\, M)\, N \xrightarrow{whr} [N/x]M}\,\mathsf{whr\_beta} \qquad \frac{M \xrightarrow{whr} M'}{M\,N \xrightarrow{whr} M'\,N}\,\mathsf{whr\_app}$$

The rules for conversion to canonical and atomic form mutually depend on each other. Note how the rules for canonical form are type-directed, while the rules for atomic form are object-directed.

$$\frac{\Gamma, x{:}A \vdash M\, x \Uparrow M' : B}{\Gamma \vdash M \Uparrow (\lambda x{:}A.\, M') : A \to B}\,\mathsf{arrow}$$

$$\frac{M \overset{whr}{\longrightarrow} M' \qquad \Gamma \vdash M' \Uparrow M'' : a}{\Gamma \vdash M \Uparrow M'' : a}\,\mathsf{whr}$$

$$\frac{\Gamma \vdash M \downarrow M' : a}{\Gamma \vdash M \Uparrow M' : a}\,\mathsf{coerce} \qquad \frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash x \downarrow x : A}\,\mathsf{var} \qquad \frac{c{:}A \text{ in } \Sigma}{\Gamma \vdash c \downarrow c : A}\,\mathsf{con}$$

$$\frac{\Gamma \vdash M \downarrow M' : A \to B \qquad \Gamma \vdash N \Uparrow N' : A}{\Gamma \vdash M\, N \downarrow M'\, N' : B}\,\mathsf{app}$$

The following properties of the simply-typed λ-calculus follow easily from known results for more conventional representations. The last is the most difficult and can be established rather elegantly using logical relations [Pfenning 2001].

6.1. THEOREM (Properties of $\lambda^{\to}$).
1. *If* $\Gamma \vdash M \Uparrow A$ *then* $\Gamma \vdash M : A$.
2. *If* $\Gamma \vdash M \downarrow A$ *then* $\Gamma \vdash M : A$.
3. *If* $\Gamma \vdash M \Uparrow M' : A$ *then* $\Gamma \vdash M' \Uparrow A$.
4. *If* $\Gamma \vdash M \downarrow M' : A$ *then* $\Gamma \vdash M' \downarrow A$.
5. *If* $\Gamma \vdash M : A$ *then there exists a unique* N *such that* $\Gamma \vdash M \Uparrow N : A$.

Two objects M and M' are definitionally equal at type A (written as $\Gamma \vdash M \equiv M' : A$) if they have the same canonical form at type A. This coincides with a notion of definitional equality based on β- and η-conversions. In particular, β- and η-conversion are admissible rules of inference to determine definitional equality of objects. We may omit the context, signature, and type and just write $M \equiv M'$. Systems are often defined based on a notion of conversion, in which case the system above could be considered as specifying an algorithm for deciding equality. The next section provides an example of this kind.

7. Appendix: the dependently typed λ-calculus

The typing rules for LF can be found under the name λP in [Barendregt and Geuvers 2001] (Chapter 18 of this Handbook), except that the rule of type conversion for LF is based on $\beta\eta$-conversion rather than just β-conversion. Because $\beta\eta$-conversion is not confluent on ill-typed terms, the standard approach to proving theoretical properties does not work in the context of LF, even though it may be adapted with some effort [Geuvers 1992, Ghani 1997, Goguen 1999].

We prefer a formulation with typed equality judgments in the style of Martin-Löf [Harper 1988] as presented in a slightly richer framework [Coquand 1991]. We call the resulting type theory λ^{Π}. First we define its basic judgments, which include typing and definitional equality. Coquand [1991] proves the correctness of an untyped algorithm for conversion which demonstrates decidability of the judgments defining LF. From this one can conclude easily that canonical (that is, long $\beta\eta$-normal) forms exist and are unique, which is critical for the adequacy theorems throughout this chapter. An alternative proof using an erasure interpretation for dependencies is given by Harper and Pfenning [2000]. We give an inductive definition of canonical forms which can be used directly in adequacy proofs to establish a compositional bijection between canonical objects of λ^{Π} and expressions or deductions in an object logic. This part is analogous to the development for the simply-typed λ-calculus in the preceding section. We also have eliminated the non-dependent function type $A \to B$ since we can think of it as an abbreviation for $\Pi x{:}A.\, B$ where x does not occur in B.

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.

Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.\, K$
Families	A	$::=$	$a \mid A\, M \mid \Pi x{:}A_1.\, A_2$
Objects	M	$::=$	$c \mid x \mid \lambda x{:}A.\, M \mid M_1\, M_2$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$

Besides the typed notion of equality, this language differs from the one given by Harper et al. [1993] in that we do not allow families to be formed by explicit abstraction. Since such families never occur in canonical forms, this does not lead to any loss in expressive power. Unlike in $\lambda^{\to}$, we can no longer introduce typing independently of definitional equality, because of the rule of type conversion motivated in Section 3.4.

$\Gamma \vdash_{\Sigma} M : A$	M has type A
$\Gamma \vdash_{\Sigma} M \equiv M' : A$	M is definitionally equal to M' at type A
$\Gamma \vdash_{\Sigma} A : K$	A has kind K
$\Gamma \vdash_{\Sigma} A \equiv A' : K$	A is definitionally equal to A' at kind K
$\Gamma \vdash_{\Sigma} K : \mathsf{kind}$	K is a valid kind
$\Gamma \vdash_{\Sigma} K \equiv K' : \mathsf{kind}$	K is definitionally equal to K'
$\vdash \Sigma\ \mathit{Sig}$	Σ is a valid signature
$\vdash_{\Sigma} \Gamma\ \mathit{Ctx}$	Γ is a valid context

These judgment are defined by the rules given below. For the typing and equality judgments we presuppose that the signature Σ and the context Γ are valid, so we do not check this in the rules for variables and constants. Furthermore, we do not have an explicit rule for η-conversion, since it, together with a congruence rule for λ-abstraction, is equivalent to the extensionality rule eq_lam for functional equality.

Valid objects

$$\frac{\Gamma \vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\, M : \Pi x{:}A.\, B}\ \mathsf{lam}$$

$$\frac{\Gamma \vdash_\Sigma A : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M\, x \equiv M'\, x : B}{\Gamma \vdash_\Sigma M \equiv M' : \Pi x{:}A.\, B}\ \mathsf{eq\_lam}$$

$$\frac{c{:}A \text{ in } \Sigma}{\Gamma \vdash_\Sigma c : A}\ \mathsf{con} \qquad \frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : A}\ \mathsf{var}$$

$$\frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.\, B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\ N : [N/x]B}\ \mathsf{app}$$

$$\frac{\Gamma \vdash_\Sigma M \equiv M' : \Pi x{:}A.\, B \qquad \Gamma \vdash_\Sigma N \equiv N' : A}{\Gamma \vdash_\Sigma M\, N \equiv M'\, N' : [N/x]B}\ \mathsf{eq\_app}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma (\lambda x{:}A.\, M)\, N \equiv [N/x]M : [N/x]B}\ \mathsf{beta}$$

Valid types

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

$$\frac{\Gamma \vdash_\Sigma A \equiv A' : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma B \equiv B' : \mathsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.\, B \equiv \Pi x{:}A'.\, B' : \mathsf{type}}\ \mathsf{eq\_pi}$$

$$\frac{a{:}K \text{ in } \Sigma}{\Gamma \vdash_\Sigma a : K}\ \mathsf{con}$$

$$\frac{\Gamma \vdash_\Sigma A : \Pi x{:}B.\, K \qquad \Gamma \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma A\ M : [M/x]K}\ \mathsf{app}$$

$$\frac{\Gamma \vdash_\Sigma A \equiv A' : \Pi x{:}B.\, K \qquad \Gamma \vdash_\Sigma M \equiv M' : B}{\Gamma \vdash_\Sigma A\, M \equiv A'\, M' : [M/x]K}\ \mathsf{eq\_app}$$

Valid kinds

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{type} : \mathsf{kind}}\ \mathsf{type}$$

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

$$\frac{\Gamma \vdash_\Sigma A \equiv A' : \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma K \equiv K' : \mathsf{kind}}{\Gamma \vdash_\Sigma \Pi x{:}A.\, K \equiv \Pi x{:}A'.\, K' : \mathsf{kind}}\ \mathsf{eq\_pi}$$

Equality rules. We present the equality rules for all three levels in abbreviated form, where U, V, and W range over objects, types, kinds, or the symbol kind as appropriate for the equality judgments shown above.

$$\frac{\Gamma \vdash_\Sigma U : V}{\Gamma \vdash_\Sigma U \equiv U : V}\ \mathsf{refl} \qquad \frac{\Gamma \vdash_\Sigma U_1 \equiv U_2 : V}{\Gamma \vdash_\Sigma U_2 \equiv U_1 : V}\ \mathsf{sym}$$

$$\frac{\Gamma \vdash_\Sigma U_1 \equiv U_2 : V \qquad \Gamma \vdash_\Sigma U_2 \equiv U_3 : V}{\Gamma \vdash_\Sigma U_1 \equiv U_3 : V}\ \mathsf{trans}$$

$$\frac{\Gamma \vdash_\Sigma U : V \qquad \Gamma \vdash_\Sigma V \equiv V' : W}{\Gamma \vdash_\Sigma U : V'}\ \mathsf{conv}$$

$$\frac{\Gamma \vdash_\Sigma U_1 \equiv U_2 : V \qquad \Gamma \vdash_\Sigma V \equiv V' : W}{\Gamma \vdash_\Sigma U_1 \equiv U_2 : V'}\ \mathsf{eq\_conv}$$

Valid signatures

$$\frac{}{\vdash \cdot\ \mathit{Sig}}\ \mathsf{sigemp} \qquad \frac{\vdash \Sigma\ \mathit{Sig} \qquad \vdash_\Sigma K : \mathsf{kind}}{\vdash \Sigma, a{:}K\ \mathit{Sig}}\ \mathsf{sigfam}$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \qquad \vdash_\Sigma A : \mathsf{type}}{\vdash \Sigma, c{:}A\ \mathit{Sig}}\ \mathsf{sigobj}$$

Valid contexts

$$\frac{}{\vdash_\Sigma \cdot\ \mathit{Ctx}}\ \mathsf{ctxemp} \qquad \frac{\vdash_\Sigma \Gamma\ \mathit{Ctx} \qquad \Gamma \vdash_\Sigma A : \mathsf{type}}{\vdash_\Sigma \Gamma, x{:}A\ \mathit{Ctx}}\ \mathsf{ctxobj}$$

We can obtain the decidability of the judgments constituting this formulation of LF via a sequence of lemmas culminating in an argument via Kripke-logical relations and an untyped algorithm for testing equality as given by Coquand [1991]. The version of this theorem for β-conversion only (where the $\mathsf{eq\_lam}$ rule is replaced by a congruence rule for λ-abstraction) is due to Harper et al. [1993].

7.1. THEOREM (Properties of LF).

1. *If* $\Gamma_1, x{:}A, y{:}B, \Gamma_2 \vdash_\Sigma M : C$ *and* $\Gamma_1 \vdash_\Sigma B : \mathsf{type}$ *then* $\Gamma_1, y{:}B, x{:}A, \Gamma_2 \vdash_\Sigma M : C$.
2. *If* $\Gamma \vdash_\Sigma M : C$ *and* $\Gamma \vdash_\Sigma A : \mathsf{type}$ *then* $\Gamma, x{:}A \vdash_\Sigma M : C$.
3. *If* $\Gamma_1, x{:}A, \Gamma_2 \vdash_\Sigma M : C$ *and* $\Gamma_1 \vdash_\Sigma N : A$ *then* $\Gamma_1, [N/x]\Gamma_2 \vdash_\Sigma [N/x]M : [N/x]C$.
4. *All judgments defining the* λ^Π *type theory are decidable.*

We single out the properties of exchange, weakening, and substitution, since they are at the core of the judgments-as-types representation technique. Note that contraction is a simple consequence of substitution in our formulation. Parametric and hypothetical judgments can be implemented as functions in λ^Π because these properties match the properties of hypotheses. Logics such as linear logic in which assumptions do not satisfy these properties must be represented with different techniques. This has led, for example, to the development of the linear logical framework [Cervesato and Pfenning 1996] which provides more control over properties of assumptions.

We continue by presenting the notions of canonical and atomic form as a judgment, generalizing the analogous judgments from the simply-typed λ-calculus in Section 6.

$\Gamma \vdash_\Sigma M \Uparrow A$	M is canonical of type A
$\Gamma \vdash_\Sigma M \downarrow A$	M is atomic of type A
$\Gamma \vdash_\Sigma A \Uparrow K$	A is canonical of kind K
$\Gamma \vdash_\Sigma A \downarrow K$	A is atomic of kind K

These judgments are defined via the following inference rules. We use P for a *base type*, that is, one which has the form $a\, M_1 \ldots M_n$ rather than $\Pi x{:}A.\, B$.

Canonical objects

$$\frac{\Gamma \vdash_\Sigma A \Uparrow \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma M \Uparrow B \qquad \Gamma \vdash_\Sigma A \equiv A' : \mathsf{type}}{\Gamma \vdash_\Sigma \lambda x{:}A.\, M \Uparrow \Pi x{:}A'.\, B}\;\mathsf{pi}$$

$$\frac{\Gamma \vdash_\Sigma M \downarrow P \qquad \Gamma \vdash_\Sigma P \equiv P' : \mathsf{type}}{\Gamma \vdash_\Sigma M \Uparrow P'}\;\mathsf{coerce}$$

Atomic objects

$$\frac{c{:}A \text{ in } \Sigma}{\Gamma \vdash_\Sigma c \downarrow A}\;\mathsf{con} \qquad \frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash_\Sigma x \downarrow A}\;\mathsf{var}$$

$$\frac{\Gamma \vdash_\Sigma M \downarrow \Pi x{:}A.\, B \qquad \Gamma \vdash_\Sigma N \Uparrow A}{\Gamma \vdash_\Sigma M\, N \downarrow [N/x]B}\;\mathsf{atmapp}$$

Canonical types

$$\frac{\Gamma \vdash_\Sigma A \Uparrow \mathsf{type} \qquad \Gamma, x{:}A \vdash_\Sigma B \Uparrow \mathsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.\, B \Uparrow \mathsf{type}}\;\mathsf{pi}$$

$$\frac{\Gamma \vdash_\Sigma P \downarrow \mathsf{type}}{\Gamma \vdash_\Sigma P \Uparrow \mathsf{type}}\;\mathsf{coerce}$$

Atomic types

$$\frac{a{:}K \text{ in } \Sigma}{\Gamma \vdash_\Sigma a \downarrow K}\;\mathsf{con}$$

$$\frac{\Gamma \vdash_\Sigma A \downarrow \Pi x{:}B.\, K \qquad \Gamma \vdash_\Sigma M \Uparrow B}{\Gamma \vdash_\Sigma A\, M \downarrow [M/x]K}\;\mathsf{app}$$

It is easy to see that canonical forms are well-typed.

7.2. THEOREM (Properties of canonical forms).

1. *If* $\Gamma \vdash_\Sigma M \Uparrow A$ *then* $\Gamma \vdash_\Sigma M : A$.
2. *If* $\Gamma \vdash_\Sigma M \downarrow A$ *then* $\Gamma \vdash_\Sigma M : A$.
3. *If* $\Gamma \vdash_\Sigma A \Uparrow K$ *then* $\Gamma \vdash_\Sigma A : K$.
4. *If* $\Gamma \vdash_\Sigma A \downarrow K$ *then* $\Gamma \vdash_\Sigma A : K$.

PROOF. By straightforward induction on the structure of the canonical and atomic forms. □

Finally we come to algorithms for conversion to canonical form. They are designed so that two terms are definitionally equal if they have the same canonical form.

$M \stackrel{whr}{\longrightarrow} M'$	M weak head reduces to M'
$\Gamma \vdash_\Sigma M \Uparrow M' : A$	M has canonical form M' at type A
$\Gamma \vdash_\Sigma M \downarrow M' : A'$	M has atomic form M' at type A'
$\Gamma \vdash_\Sigma A \Uparrow A' : K$	A has canonical form A' at kind K
$\Gamma \vdash_\Sigma A \downarrow A' : K'$	A has atomic form A' at kind K'

To read these judgments as algorithms we apply the logic programming interpretation of these rules for the bottom-up construction of a derivation. In weak head reduction we assume that M is given and M' is constructed. In the judgments for conversion to canonical form we assume that Σ, Γ, M, A, and K are given while we construct M' and A'. In the judgments for atomic forms we assume Σ, Γ, M, and A to be given and construct M', A' and K'.

Weak head reduction

$$\frac{}{(\lambda x{:}A.\,M)\;N \stackrel{whr}{\longrightarrow} [N/x]M}\;\textsf{whr\_beta}$$

$$\frac{M \stackrel{whr}{\longrightarrow} M'}{M\;N \stackrel{whr}{\longrightarrow} M'\;N}\;\textsf{whr\_app}$$

Conversion to canonical objects

$$\frac{\Gamma \vdash_\Sigma A \Uparrow A' : \textsf{type} \qquad \Gamma, x{:}A' \vdash_\Sigma M\;x \Uparrow M' : B}{\Gamma \vdash_\Sigma M \Uparrow \lambda x{:}A'.\,M' : \Pi x{:}A.\,B}\;\textsf{pi}$$

$$\frac{\Gamma \vdash_\Sigma M \downarrow M' : P \qquad \Gamma \vdash_\Sigma P \equiv P'}{\Gamma \vdash_\Sigma M \Uparrow M' : P'}\;\textsf{atm}$$

$$\frac{M \stackrel{whr}{\longrightarrow} M' \qquad \Gamma \vdash_\Sigma M' \Uparrow M'' : P}{\Gamma \vdash_\Sigma M \Uparrow M'' : P}\;\textsf{whr}$$

Conversion to atomic objects

$$\frac{c{:}A \text{ in } \Sigma}{\Gamma \vdash_\Sigma c \downarrow c : A}\;\textsf{con} \qquad \frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash_\Sigma x \downarrow x : A}\;\textsf{var}$$

$$\frac{\Gamma \vdash_\Sigma M \downarrow M' : \Pi x{:}A.\,B \qquad \Gamma \vdash_\Sigma N \Uparrow N' : A}{\Gamma \vdash_\Sigma M\;N \downarrow M'\;N' : [M'/x]B}\;\textsf{app}$$

Conversion to canonical types

$$\frac{\Gamma \vdash_\Sigma A \Uparrow A' : \textsf{type} \qquad \Gamma, x{:}A' \vdash_\Sigma B \Uparrow B' : \textsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.\,B \Uparrow \Pi x{:}A'.\,B' : \textsf{type}}\;\textsf{pi}$$

$$\frac{\Gamma \vdash_\Sigma P \downarrow P' : \textsf{type}}{\Gamma \vdash_\Sigma P \Uparrow P' : \textsf{type}}\;\textsf{atm}$$

Conversion to atomic types

$$\frac{a{:}K \text{ in } \Sigma}{\Gamma \vdash_\Sigma a \downarrow a : K}\;\textsf{con}$$

$$\frac{\Gamma \vdash_\Sigma A \downarrow A' : \Pi x{:}B.\,K \qquad \Gamma \vdash_\Sigma M \Uparrow M' : B}{\Gamma \vdash_\Sigma A\;M \downarrow A'\;M' : [M'/x]K}\;\textsf{app}$$

We show only the relevant properties for canonical forms on objects—atomic forms, types, and kinds satisfy similar properties.

7.3. Theorem (Convertibility).

1. *If* $\Gamma \vdash_\Sigma M \Uparrow M' : A$ *then* $\Gamma \vdash_\Sigma M' \Uparrow A$.
2. *If* $\Gamma \vdash_\Sigma M \Uparrow M' : A$ *then* $\Gamma \vdash_\Sigma M \equiv M' : A$.
3. *If* $\Gamma \vdash_\Sigma M : A$ *then there is a unique* M' *such that* $\Gamma \vdash_\Sigma M \Uparrow M' : A$.
4. $\Gamma \vdash_\Sigma M \equiv M' : A$ *iff* $\Gamma \vdash_\Sigma M \Uparrow N : A$ *and* $\Gamma \vdash_\Sigma M' \Uparrow N : A$ *for some* N.

Proof. The first two properties follow by simple structural inductions. The last two follow from Coquand's algorithm [Coquand 1991] by additional η-expansions. Related proofs are given by Harper and Pfenning [2000] and Virga [1999]. □

8. 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.

Equational vs. deductive encodings. Logical frameworks based on rewriting logic [Martì-Oliet and Meseguer 1993] (variations of which are implemented in Maude [*Maude* 1999] and ELAN [*ELAN* 1998, Kirchner et al. 1993, Haberstrau 1994, Borovanský et al. 1998]) are based on equational reasoning, rewriting, and constraints, while others discussed in this chapter (LF, hereditary Harrop formulas, FS_0, ALF) are based on deductive reasoning. It is clear that each approach can be simulated in the other, but usually with some loss of clarity, efficiency and elegance for certain classes of applications. Rewriting logic, for example, deals particularly well with concurrency, while it does not seem well suited for situations where deductions themselves need to be reified in the meta-language. First steps for combining ideas from these classes of frameworks are the rewriting mechanisms in Isabelle [Nipkow 1989] and the study of term rewriting in higher-order languages with dependent types [Virga 1996, Virga 1999]. For more on rewriting logic and its use as a logical framework, see [Meseguer 1998, Kirchner and Kirchner 1998]. The semantic origin of this work is *institutions* [Goguen and Burstall 1992]; a connection is made by Meseguer [1987].

Strong vs. weak frameworks. De Bruijn, the founder of the field of logical frameworks, argues in [de Bruijn 1991*a*] 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. While most frameworks are based on weak fragments of intuitionistic logic or type theory, *labelled deductive systems* as proposed by Gabbay [1994, 1996] are a notable exception. They are based essentially on classical, first-order logic where deductions are restricted through the use of labels endowed with an equational theory. Proof search can proceed, for example, by classical resolution techniques. For more on this approach, see [Ohlbach et al. 2001] (Chapter 21 of this Handbook). This encoding is well-suited for modal logics, but it appears less immediately applicable to other deductive systems, especially those arising in the theory of programming languages.

Inductive representations vs. higher-order abstract syntax. This is related to the previous question. Inductive representations of logics are supported in FS_0 [Feferman 1988] and ALF [Magnusson and Nordström 1994] and many logics not explicitly designed as logical frameworks such as Nuprl [Basin and Constable 1993], LEGO [Pollack 1994], Coq [Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring and Werner 1993], and Isabelle/HOL [Paulson 1993]. 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. These are the foundation of LF (underlying Elf) and hereditary Harrop formulas (underlying λProlog and Isabelle). Present work on combining advantages of both either employ reflection [Despeyroux et al. 1997, Leleu 1998] or formal meta-reasoning about the logical framework itself [McDowell and Miller 1997, Schürmann and Pfenning 1998, 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 (see Section 8.2).

Functional vs. logical meta-programming. ML has originally been designed as a meta-language to program theorem provers for complex logics. It is still used in this capacity in many theorem proving environments and logical frameworks, including Isabelle. The strategy language of ELAN is similar, but has rich primitives for non-deterministic search which have to be programmed in ML, a sequential language. The functional meta-language approach has the disadvantage that the programmer must deal with many languages: the object logic, the logical framework, and the implementation language of the logical framework. A more uniform approach is to directly give an operational semantics to the logical framework in the spirit of abstract logic programming [Miller et al. 1991]. This makes it quite easy to program algorithms, but this approach has some drawbacks when it comes to user interaction.

8.1. Framework extensions

Logical framework languages are judged 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, some of them have even led to explicit negative results [Gardner 1992]. We briefly summarize some of the current research on refining or extending logical frameworks. Any proposed extension must carefully weigh the benefits for classes of applications against the complications it introduces into the meta-theory.

Substructural extensions. Frameworks such as hereditary Harrop formulas or LF can encode linear and other substructural logics [Girard 1987], but their encodings are not as direct as one might hope. The reason is that linear assumptions (each of which must be used exactly once) can not be modeled as hypotheses in the meta-language (which satisfy weakening and contraction). For similar reasons, the store in the encoding of an imperative programming language cannot be modeled via hypotheses on the values of the cells in the store. The linear frameworks Forum and linear LF have been designed to overcome these limitations. Forum [Miller 1994] is based on classical linear logic and extends hereditary Harrop formulas. Chirimar [1995] shows how to apply Forum to the theory of imperative programming languages. Linear LF [Cervesato and Pfenning 1997] is a conservative extension of LF with linear hypotheses. The desirable properties of LF are retained when the new connectives are restricted to linear implication, additive conjunction, and additive truth. Unlike Forum, the connectives are interpreted intuitionistically, which allows proof terms with decidable equality and type-checking relations to reify linear deductions and imperative computations. Applications to imperative programming can be found in [Cervesato 1996], applications to cut-elimination in both classical and intuitionistic sequent calculi are given in [Pfenning 1994*b*].

Subtyping. In many cases an object language or logic exhibits natural subtyping relationships. For example, deductions in normal form may be considered a subtype of arbitrary natural deductions. In the absence of subtyping, these can be coded either as explicit higher-level judgments or via explicit coercions, in both cases often significantly complicating the representation. In [Pfenning 1993], we have proposed an extension of LF to permit a simple and decidable subtyping judgment. Despite its relative simplicity it complicates unification and proof search [Kohlhase and Pfenning 1993] and the pragmatic consequences are unclear at present. Other approaches for general type theories have also been proposed recently [Aspinall and Compagnoni 1996], but their practicality in the context of logical frameworks is untested.

Polymorphism. Both Isabelle and λProlog allow polymorphism in the presentation of logics; in the case of Isabelle this includes sort restrictions on type variables. Like subtyping, polymorphism significantly complicates unification and proof search. Adequacy of encodings using higher-order abstract syntax is also more difficult to prove, since the notion of η-long form is more complex [Dowek, Huet and Werner 1993, Ghani 1997] and not preserved under substitution for type variables. On the other hand, polymorphism avoids code duplication—a similar effect might be achieved with module systems instead.

Module languages. The modular presentation of logical systems has always been considered important. For Automath, de Bruijn has proposed the notion of telescope [de Bruijn 1991*b*] as a modularity mechanism. For pure type systems [Barendregt 1992] (which include λ^Π as a subcalculus) Courant [1997, 1999] has described a general module calculus. The modular presentation of logics has been investigated in [Harper, Sannella and Tarlecki 1989*a*, Harper, Sannella and Tarlecki 1989*b*, Harper, Sannella and Tarlecki 1994] and cast in a concrete module language for Elf in [Harper and Pfenning 1998] following the ideas of signatures and functors in ML. Rewriting logic also explicitly supports logic morphisms within a flexible module language based on [Meseguer 1987]. The notion of theory in Isabelle provides another structuring mechanism [Nipkow 1993]. The module language for λProlog is more concerned with the operational semantics and search spaces while remaining based on solid logical foundations [Miller 1986, Miller 1989, Nadathur and Tong 1999].

8.2. *Proof-carrying code*

An important recent application of logical frameworks is the notion of *proof-carrying code* (PCC) [Necula 1997] and certifying compilation [Necula 1998, Necula and Lee 1998*a*]. Proof-carrying code is a safety infrastructure for mobile code and operating system extension. A code producer supplies not only a binary executable but also a proof of its safety according to some predetermined safety policy. This proof is

expressed as an object in the LF logical framework, although other type-theoretic frameworks could be used as well. The code consumer downloads the binary and proof object and checks the safety proof against the binary. This is accomplished by generating a verification condition A from the binary in a single, linear sweep and then checking the proof object M against the verification condition by simple LF type-checking, $M : A$.

A safety policy is expressed by a verification condition generator and an LF signature which encodes the proof rules for verification conditions. Examples of such safety policies are type safety and memory safety, guaranteeing that a program will not access memory outside its address space [Necula 1998]. Another example is resource bounds in operating systems extensions such as packet filters [Necula and Lee 1996].

Since both the verification condition generator and the LF type-checker are relatively small (compared to compilers or theorem provers), the trusted computing base of this architecture is quite small. The use of a logical framework where deductions are reified as objects allows one single implementation to support multiple safety policies and proof rules, increasing trust in the reliability of the architecture, especially since the properties of LF are well understood and thoroughly investigated.

The realization of proof-carrying code raised some interesting directions for the development of logical frameworks. Here we consider two: how do we generate proof objects and how can we eliminate redundancy from LF objects to achieve compact encodings of proofs?

The generation of proof objects is the task of a *certifying compiler* which takes advantage of properties of the source language to generate annotations on the assembly code. In case of the Touchstone compiler [Necula 1998], this is a safe subset of C. The annotations guarantee that a specialized theorem prover has enough information to derive the verification condition for the binary. The specialized theorem prover maintains enough information to generate LF proof objects with respect to the axioms and inference rules available for the given safety policy. For type and memory safety, this has been shown to be practical, including a proof-generating version of the simplex algorithm described in [Necula 1998]. Thus, the theorem prover as a whole does not need to be trusted, since it generates derivations which can be verified independently.

The second question concerns the elimination of redundancy in the LF representation of derivations. A first proposal in this direction for the Elf logic programming language was made in [Michaylov and Pfenning 1992]. In PCC, the representation can be further optimized [Necula and Lee 1998*b*] since the main operation we are concerned with is type-checking, while Elf has to support unification and proof search. The principle, however is the same and goes back to the notion of strictness in functional languages. This has been analyzed by Pfenning and Schürmann [1998*a*].

8.3. 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. A survey with deeper coverage of modal logics and inductive definitions can be found in [Basin and Matthews 2000]. The textbook [Pfenning 2001] provides a gentler and more thorough introduction to the pragmatics of the LF logical framework and its use for the study of programming languages. The author also maintains a home page on logical frameworks [*Logical Frameworks* 1994] at `http://www.cs.cmu.edu/~fp/lfs.html` which is periodically updated, and which contains a more extensive bibliography and pointers to implementations, mailing lists, and related material.

Bibliography

ABEL A. [1999], A semantic analysis of structural recursion, Master's thesis, Ludwig-Maximilians-Universität München.

ALLEN S. F., CONSTABLE R. L., HOWE D. J. AND AITKEN W. E. [1990], The semantics of reflected proof, *in* 'Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS'90)', IEEE Computer Society Press, pp. 95–105.

ALTENKIRCH T., GASPES V., NORDSTRÖM B. AND VON SYDOW B. [1994], *A User's Guide to ALF*, Chalmers University of Technology, Sweden.

ANDREOLI J.-M. [1992], 'Logic programming with focusing proofs in linear logic', *Journal of Logic and Computation* **2**(3), 297–347.

ANDREWS P. [2001], Classical type theory, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 15, pp. 965–1007.

ASPINALL D. AND COMPAGNONI A. [1996], Subtyping dependent types, *in* E. Clarke, ed., 'Proceedings of the 11th Annual Symposium on Logic in Computer Science', IEEE Computer Society Press, New Brunswick, New Jersey, pp. 86–97.

BARENDREGT H. AND GEUVERS H. [2001], Proof-assistants using dependent type systems, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 18, pp. 1149–1238.

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

BARENDREGT H. P. [1992], Lambda calculi with types, *in* S. Abramsky, D. Gabbay and T. Maibaum, eds, 'Handbook of Logic in Computer Science', Vol. 2, Oxford University Press, chapter 2, pp. 117–309.

BASIN D. A. AND CONSTABLE R. L. [1993], Metalogical frameworks, *in* G. Huet and G. Plotkin, eds, 'Logical Environments', Cambridge University Press, pp. 1–29.

BASIN D. AND MATTHEWS S. [1996], Structuring metatheory on inductive definitions, *in* M. McRobbie and J. Slaney, eds, 'Proceedings of the 13th International Conference on Automated Deduction (CADE-13)', Springer-Verlag LNAI 1104, New Brunswick, New Jersey, pp. 171–185.

BASIN D. AND MATTHEWS S. [2000], Logical frameworks, *in* D. Gabbay and F. Guenthner, eds, 'Handbook of Philosophical Logic', 2nd edn, Kluwer Academic Publishers. In preparation.

BASIN D., MATTHEWS S. AND VIGANÒ L. [1998], A modular presentation of modal logics in a logical framework, *in* 'The Tbilisi Symposium on Language, Logic and Computation: Selected Papers', CSLI Publications.

BOROVANSKÝ P., KIRCHNER C., KIRCHNER H., MOREAU P.-E. AND RINGEISSEN C. [1998], An overview of ELAN, *in* C. Kirchner and H. Kirchner, eds, 'Proceedings of the International Workshop on Rewriting Logic and its Applications', Vol. 15 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Pont-à-Mousson, France.
URL: *http://www.elsevier.com/locate/entcs/volume15.html*

CERVESATO I. [1996], A Linear Logical Framework, PhD thesis, Dipartimento di Informatica, Università di Torino.

CERVESATO I. AND PFENNING F. [1996], A linear logical framework, *in* E. Clarke, ed., 'Proceedings of the Eleventh Annual Symposium on Logic in Computer Science', IEEE Computer Society Press, New Brunswick, New Jersey, pp. 264–275.

CERVESATO I. AND PFENNING F. [1997], Linear higher-order pre-unification, *in* G. Winskel, ed., 'Proceedings of the Twelfth Annual Sumposium on Logic in Computer Science (LICS'97)', IEEE Computer Society Press, Warsaw, Poland, pp. 422–433.

CHIRIMAR J. L. [1995], Proof Theoretic Approach to Specification Languages, PhD thesis, University of Pennsylvania.

CHURCH A. AND ROSSER J. [1936], 'Some properties of conversion', *Transactions of the American Mathematical Society* **39**(3), 472–482.

CONSTABLE R. L. ET AL. [1986], *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, New Jersey.

Coq [1999], Project home page. Version 6.2.3.
URL: *http://pauillac.inria.fr/coq/*

COQUAND C. [1992], A proof of normalization for simply typed lambda calculus written in ALF, *in* 'Proceedings of the Workshop on Types for Proofs and Programs', Båstad, Sweden, pp. 85–92.

COQUAND T. [1991], An algorithm for testing conversion in type theory, *in* G. Huet and G. Plotkin, eds, 'Logical Frameworks', Cambridge University Press, pp. 255–279.

COQUAND T., NORDSTRÖM B., SMITH J. M. AND VON SYDOW B. [1994], 'Type theory and programming', *Bulletin of the European Association for Theoretical Computer Science* **52**, 203–228.

COQUAND T. AND SMITH J. M. [1993], What is the status of pattern matching in type theory?, *in* 'Proceedings of the Workshop on Types for Proofs and Programs', Nijmegen, The Netherlands, pp. 91–94.

COURANT J. [1997], A module calculus for pure type systems, *in* P. de Groote and R. Hindley, eds, 'Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)', Springer-Verlag LNCS, Nancy, France, pp. 112–128.

COURANT J. [1999], MC: a modular calculus for Pure Type Systems, Rapport de Recherche 1217, CNRS Université Paris Sud.

CURRY H. B. AND FEYS R. [1958], *Combinatory Logic*, North-Holland, Amsterdam.

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', Springer-Verlag LNM 125, Versailles, France, pp. 29–61.

DE BRUIJN N. [1972], 'Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem', *Indag. Math.* **34**(5), 381–392.

DE BRUIJN N. [1980], A survey of the project AUTOMATH, *in* J. Seldin and J. Hindley, eds, 'To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism', Academic Press, pp. 579–606.

DE BRUIJN N. [1991*a*], A plea for weaker frameworks, *in* G. Huet and G. Plotkin, eds, 'Logical Frameworks', Cambridge University Press, pp. 40–67.

de Bruijn N. [1991*b*], 'Telescopic mappings in typed lambda calculus', *Information and Computation* **91**(2), 189–204.

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.

Degtyarev A. and Voronkov A. [2001], The inverse method, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. I, Elsevier Science, chapter 4, pp. 179–272.

Despeyroux J., Felty A. and Hirschowitz A. [1995], Higher-order abstract syntax in Coq, *in* M. Dezani-Ciancaglini and G. Plotkin, eds, 'Proceedings of the International Conference on Typed Lambda Calculi and Applications', Springer-Verlag LNCS 902, Edinburgh, Scotland, pp. 124–138.

Despeyroux J. and Hirschowitz A. [1994], Higher-order abstract syntax with induction in Coq, *in* F. Pfenning, ed., 'Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning', Springer-Verlag LNAI 822, Kiev, Ukraine, pp. 159–173.

Despeyroux J., Pfenning F. and Schürmann C. [1997], Primitive recursion for higher-order abstract syntax, *in* R. Hindley, ed., 'Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)', Springer-Verlag LNCS 1210, Nancy, France, pp. 147–163. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

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', Springer-Verlag LNCS 664, Utrecht, The Netherlands, pp. 139–145.

Dowek G. [2001], Higher-order unification and matching, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 16, pp. 1009–1062.

Dowek G., Felty A., Herbelin H., Huet G., Murthy C., Parent C., Paulin-Mohring C. and Werner B. [1993], The Coq proof assistant user's guide, Rapport Techniques 154, INRIA, Rocquencourt, France. Version 5.8.

Dowek G., Hardin T., Kirchner C. and Pfenning F. [1996], Unification via explicit substitutions: The case of higher-order patterns, *in* M. Maher, ed., 'Proceedings of the Joint International Conference and Symposium on Logic Programming', MIT Press, Bonn, Germany, pp. 259–273.

Dowek G., Huet G. and Werner B. [1993], On the definition of the eta-long normal form in type systems of the cube, *in* H. Geuvers, ed., 'Informal Proceedings of the Workshop on Types for Proofs and Programs', Nijmegen, The Netherlands.

Dyckhoff R. and Pinto L. [1994], Uniform proofs and natural deduction, *in* D. Galmiche and L. Wallen, eds, 'Proceedings of the Workshop on Proof Search in Type-Theoretic Languages', Nancy, France, pp. 17–23.

ELAN [1998], System home page. Version 3.3.
URL: *http://www.loria.fr/ELAN*

Elliott C. [1989], Higher-order unification with dependent types, *in* N. Dershowitz, ed., 'Rewriting Techniques and Applications', Springer-Verlag LNCS 355, Chapel Hill, North Carolina, pp. 121–136.

Elliott C. M. [1990], Extensions and Applications of Higher-Order Unification, PhD thesis, School of Computer Science, Carnegie Mellon University. Available as Technical Report CMU-CS-90-134.

Eriksson L.-H. [1992], A finitary version of the calculus of partial inductive definitions, *in* L.-H. Eriksson, L. Hallnäs and P. Schroeder-Heister, eds, 'Proceedings of the Second International Workshop on Extensions of Logic Programming', Springer-Verlag LNAI 596, Stockholm, Sweden, pp. 89–134.

Eriksson L.-H. [1993*a*], Finitary Partial Inductive Definitions and General Logic, PhD thesis, Department of Computer and System Sciences, Royal Institute of Technology, Stockholm.

Eriksson L.-H. [1993*b*], Finitary partial inductive definitions as a general logic, *in* R. Dyckhoff, ed., 'Proceedings of the 4th International Workshop on Extensions of Logic Programming', Springer-Verlag LNAI 798.

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', Springer Verlag LNAI 814, Nancy, France, pp. 821–825.

Feferman S. [1988], Finitary inductive systems, *in* R. Ferro, ed., 'Proceedings of Logic Colloquium '88', North-Holland, Padova, Italy, pp. 191–220.

Felty A. [1989], Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language, PhD thesis, University of Pennsylvania. Available as Technical Report MS-CIS-89-53.

Felty A. [1993], 'Implementing tactics and tacticals in a higher-order logic programming language', *Journal of Automated Reasoning* **11**(1), 43–81.

Felty A. and Miller D. [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', Springer-Verlag LNCS 310, Argonne, Illinois, pp. 61–80.

Felty A. and Miller D. [1990], Encoding a dependent-type λ-calculus in a logic programming language, *in* M. Stickel, ed., '10th International Conference on Automated Deduction', Springer-Verlag LNCS 449, Kaiserslautern, Germany, pp. 221–235.

Frege G. [1879], *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*, Verlag von Louis Nebert. English translation *Begriffsschrift, a formula language, modeled upon that of arithmatic, for pure thought* in J. van Heijenoort, editor, *From Frege to Gödel; A Source Book in Mathematical Logic, 1879–1931*, pp. 1–82, Harvard University Press, 1967.

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, chapter 2.6.

Gabbay D. M. [1996], *Labelled Deductive Systems*, Vol. 1, Oxford University Press.

Gardner P. [1992], Representing Logics in Type Theory, PhD thesis, University of Edinburgh. Available as Technical Report CST-93-92.

Gaspes V. and Smith J. M. [1992], Machine checked normalization proofs for typed combinator calculi, *in* 'Proceedings of the Workshop on Types for Proofs and Programs', Båstad, Sweden, pp. 177–192.

Gentzen G. [1935], 'Untersuchungen über das logische Schließen', *Mathematische Zeitschrift* **39**, 176–210, 405–431. English translation *Investigations into logical deductions* in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pp. 68–131, North-Holland Publishing Co., 1969.

Geuvers H. [1992], The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi, *in* A. Scedrov, ed., 'Seventh Annual IEEE Symposium on Logic in Computer Science', Santa Cruz, California, pp. 453–460.

Ghani N. [1997], Eta-expansions in dependent type theory — the calculus of constructions, *in* P. de Groote and J. Hindley, eds, 'Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)', Springer-Verlag LNCS 1210, Nancy, France, pp. 164–180.

Girard J.-Y. [1987], 'Linear logic', *Theoretical Computer Science* **50**, 1–102.

Girard J.-Y. [1993], 'On the unity of logic', *Annals of Pure and Applied Logic* **59**, 201–217.

Goguen H. [1999], Soundness of the logical framework for its typed operational semantics, *in* J.-Y. Girard, ed., 'Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)', Springer-Verlag LNCS 1581, L'Aquila, Italy, pp. 177–197.

Goguen J. A. and Burstall R. M. [1992], 'Institutions: Abstract model theory for specification and programming', *Journal of the ACM* **39**(1), 95–146.

Goldfarb W. D. [1981], 'The undecidability of the second-order unification problem', *Theoretical Computer Science* **13**, 225–230.

Gordon M. J., Milner R. and Wadsworth C. P. [1979], *Edinburgh LCF*, Springer-Verlag LNCS 78.

Gordon M. and Melham T. [1993], *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press.

Haberstrau M. [1994], ECOLOG: An environment for constraint logics, *in* J.-P. Jouannaud, ed., 'Proceedings of the First International Conference on Constraints in Computational Logics', Springer-Verlag LNCS 845, Munich, Germany, pp. 237–252.

Hallnäs L. [1987], A note on the logic of a logic program, *in* 'Proceedings of the Workshop on Programming Logic', University of Göteborg and Chalmers University of Technology, Report PMG-R37.

Hallnäs L. [1991], 'Partial inductive definitions', *Theoretical Computer Science* **87**(1), 115–142.

Hannan J. [1993], 'Extended natural semantics', *Journal of Functional Programming* **3**(2), 123–152.

Hannan J. J. [1991], Investigating a Proof-Theoretic Meta-Language for Functional Programs, PhD thesis, University of Pennsylvania. Available as Technical Report MS-CIS-91-09.

Harper R. [1988], An equational formulation of LF, Technical Report ECS-LFCS-88-67, University of Edinburgh.

Harper R., Honsell F. and Plotkin G. [1987], A framework for defining logics, *in* 'Symposium on Logic in Computer Science', IEEE Computer Society Press, pp. 194–204.

Harper R., Honsell F. and Plotkin G. [1993], 'A framework for defining logics', *Journal of the Association for Computing Machinery* **40**(1), 143–184.

Harper R. and Pfenning F. [1998], 'A module system for a programming language based on the LF logical framework', *Journal of Logic and Computation* **8**(1), 5–31.

Harper R. and Pfenning F. [2000], On equivalence and canonical forms in the LF type theory, Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University.

Harper R., Sannella D. and Tarlecki A. [1989*a*], Logic representation, *in* D. Pitt, D. Rydeheard, P. Dybjer, A. Pitts and A. Poigneé, eds, 'Proceedings of the Workshop on Category Theory and Computer Science', Springer-Verlag LNCS 389, Manchester, UK, pp. 250–272.

Harper R., Sannella D. and Tarlecki A. [1989*b*], Structure and representation in LF, *in* 'Fourth Annual Symposium on Logic in Computer Science', IEEE Computer Society Press, Pacific Grove, California, pp. 226–237.

Harper R., Sannella D. and Tarlecki A. [1994], 'Structured presentations and logic representations', *Annals of Pure and Applied Logic* **67**, 113–160.

Hayashi S. and Nakano H. [1988], *PX: A Computational Logic*, Foundations of Computing Series, MIT Press.

Hilbert D. and Bernays P. [1934], *Grundlagen der Mathematik*, Springer-Verlag, Berlin.

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.

Huet G. [1973], 'The undecidability of unification in third order logic', *Information and Control* **22**(3), 257–267.

Huet G. [1975], 'A unification algorithm for typed λ-calculus', *Theoretical Computer Science* **1**, 27–57.

Huet G. and Lang B. [1978], 'Proving and applying program transformations expressed with second-order patterns', *Acta Informatica* **11**, 31–55.

Isabelle [1998], System home page. Version 98-1.
URL: *http://www.cl.cam.ac.uk/Research/HVG/Isabelle/*

Jutting L. [1977], Checking Landau's "Grundlagen" in the AUTOMATH System, PhD thesis, Eindhoven University of Technology.

Kahn G. [1987], Natural semantics, *in* 'Proceedings of the Symposium on Theoretical Aspects of Computer Science', Springer-Verlag LNCS 247, pp. 22–39.

Kirchner, C. and Kirchner, H., eds [1998], *Proceedings of the International Workshop on Rewriting Logic and its Applications*, Vol. 15 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Pont-à-Mousson, France.
URL: *http://www.elsevier.com/locate/entcs/volume15.html*

Kirchner C., Kirchner H. and Vittek M. [1993], Implementing computational systems with constraints, *in* P. van Hentenryck and V. Saraswat, eds, 'Proceedings of the First Workshop on Principles and Practice of Constraints Programming', MIT Press, Newport, Rhode Island.

Kohlhase M. and Pfenning F. [1993], Unification in a λ-calculus with intersection types, *in* D. Miller, ed., 'Proceedings of the International Logic Programming Symposium', MIT Press, Vancouver, Canada, pp. 488–505.

Lambek J. and Scott P. [1986], *Introduction to Higher-Order Categorical Logic*, Cambridge University Press.

LEGO [1998], System home page. Version 1.3.1.
URL: *http://www.dcs.ed.ac.uk/home/lego*

Leleu P. [1998], Induction et Syntaxe Abstraite d'Ordre Supérieur dans les Théories Typées, PhD thesis, Ecole Nationale des Ponts et Chaussees, Marne-la-Vallee, France.

Logical Frameworks [1994], Home page. Includes bibliography and pointers to implementations. Last updated June 1997.
URL: *http://www.cs.cmu.edu/~fp/lfs.html*

Luo Z. and Pollack R. [1992], The LEGO proof development system: A user's manual, Technical Report ECS-LFCS-92-211, University of Edinburgh.

Magnusson L. [1993], Refinement and local undo in the interactive proof editor ALF, *in* 'Proceedings of the Workshop on Types for Proofs and Programs', Nijmegen, The Netherlands, pp. 191–208.

Magnusson L. [1995], The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution, PhD thesis, Chalmers University of Technology and Göteborg University.

Magnusson L. and Nordström B. [1994], The ALF proof editor and its proof engine, *in* H. Barendregt and T. Nipkow, eds, 'Types for Proofs and Programs', Springer-Verlag LNCS 806, pp. 213–237.

Martì-Oliet N. and Meseguer J. [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', North-Holland, pp. 153–175.

Martin-Löf P. [1985*a*], 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.

Martin-Löf P. [1985*b*], Truth of a proposition, evidence of a judgement, validity of a proof. Notes to a talk given at the workshop *Theory of Meaning*, Centro Fiorentino di Storia e Filosofia della Scienza.

Matthews S., Smaill A. and Basin D. [1993], Experience with FS_0 as a framework theory, *in* G. Huet and G. Plotkin, eds, 'Logical Environments', Cambridge University Press, pp. 61–82.

Maude [1999], System home page. Version 1.00.
URL: *http://maude.csl.sri.com*

McDowell R. [1997], Reasoning in a Logic with Definitions and Induction, PhD thesis, University of Pennsylvania.

McDowell R. and Miller D. [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', IEEE Computer Society Press, Warsaw, Poland, pp. 434–445.

Meseguer J. [1987], General logics, *in* H.-D. Ebbinghaus, ed., 'Logic Colloquium '87', North-Holland, Granada, Spain, pp. 275–329.

Meseguer, J., ed. [1998], *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, Vol. 4 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Pacific Grove, California.
URL: *http://www.elsevier.com/locate/entcs/volume4.html*

Michaylov S. and Pfenning F. [1991], Natural semantics and some of its meta-theory in Elf, *in* L.-H. Eriksson, L. Hallnäs and P. Schroeder-Heister, eds, 'Proceedings of the Second International Workshop on Extensions of Logic Programming', Springer-Verlag LNAI 596, Stockholm, Sweden, pp. 299–344.

Michaylov S. and Pfenning F. [1992], An empirical study of the runtime behavior of higher-order logic programs, *in* D. Miller, ed., 'Proceedings of the Workshop on the λProlog Programming Language', University of Pennsylvania, Philadelphia, Pennsylvania, pp. 257–271. Available as Technical Report MS-CIS-92-86.

Michaylov S. and Pfenning F. [1993], Higher-order logic programming as constraint logic programming, *in* 'Position Papers for the First Workshop on Principles and Practice of Constraint Programming', Brown University, Newport, Rhode Island, pp. 221–229.

Miller D. [1986], A theory of modules for logic programming, *in* R. M. Keller, ed., 'Third Annual IEEE Symposium on Logic Programming', Salt Lake City, Utah, pp. 106–114.

Miller D. [1989], 'A logical analysis of modules in logic programming', *Journal of Logic Programming* **6**(1-2), 79–108.

Miller D. [1991], 'A logic programming language with lambda-abstraction, function variables, and simple unification', *Journal of Logic and Computation* **1**(4), 497–536.

Miller D. [1994], A multiple-conclusion meta-logic, *in* S. Abramsky, ed., 'Ninth Annual Symposium on Logic in Computer Science', IEEE Computer Society Press, Paris, France, pp. 272–281.

Miller D., Nadathur G., Pfenning F. and Scedrov A. [1991], 'Uniform proofs as a foundation for logic programming', *Annals of Pure and Applied Logic* **51**, 125–157.

Nadathur G. and Miller D. [1988], An overview of λProlog, *in* K. A. Bowen and R. A. Kowalski, eds, 'Fifth International Logic Programming Conference', MIT Press, Seattle, Washington, pp. 810–827.

Nadathur G. and Mitchell D. J. [1999], System description: Teyjus—a compiler and abstract machine based implementation of lambda Prolog, *in* H. Ganzinger, ed., 'Proceedings of the 16th International Conference on Automated Deduction (CADE-16)', Springer-Verlag LNCS, Trento, Italy, pp. 287–291.

Nadathur G. and Tong G. [1999], 'Realizing modularity in lambdaProlog', *Journal of Functional and Logic Programming* **1999**(9).

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)', ACM Press, Paris, France, pp. 106–119.

Necula G. C. [1998], Compiling with Proofs, PhD thesis, Carnegie Mellon University. Available as Technical Report CMU-CS-98-154.

Necula G. C. and Lee P. [1996], Safe kernel extensions without run-time checking, *in* 'Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)', Seattle, Washington, pp. 229–243.

Necula G. C. and Lee P. [1998*a*], The design and implementation of a certifying compiler, *in* K. D. Cooper, ed., 'Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)', ACM Press, Montreal, Canada, pp. 333–344.

Necula G. C. and Lee P. [1998*b*], Efficient representation and validation of logical proofs, *in* V. Pratt, ed., 'Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)', IEEE Computer Society Press, Indianapolis, Indiana, pp. 93–104.

Nederpelt, R., Geuvers, J. and de Vrijer, R., eds [1994], *Selected Papers on Automath*, Vol. 133 of *Studies in Logic and the Foundations of Mathematics*, North-Holland.

Nipkow T. [1989], 'Equational reasoning in Isabelle', *Science of Computer Programming* **12**, 123–149.

Nipkow T. [1993], Order-sorted polymorphism in Isabelle, *in* G. Huet and G. Plotkin, eds, 'Logical Environments', Cambridge University Press, pp. 164–188.

Nipkow T. and Paulson L. C. [1992], Isabelle-91, *in* D. Kapur, ed., 'Proceedings of the 11th International Conference on Automated Deduction', Springer-Verlag LNAI 607, Saratoga Springs, NY, pp. 673–676. System abstract.

Nordström B. [1993], The ALF proof editor, *in* 'Proceedings of the Workshop on Types for Proofs and Programs', Nijmegen, pp. 253–266.

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

Nuprl [1999], Project home page. Version 4.2.
URL: *http://simon.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html*

Ohlbach H., Nonnengart A., de Rijke M. and Gabbay D. [2001], Encoding two-valued nonclassical logics in classical logic, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 21, pp. 1403–1486.

Paulin-Mohring C. [1993], Inductive definitions in the system Coq: Rules and properties, *in* M. Bezem and J. Groote, eds, 'Proceedings of the International Conference on Typed Lambda Calculi and Applications', Springer-Verlag LNCS 664, Utrecht, The Netherlands, pp. 328–345.

Paulson L. [1983], Tactics and tacticals in Cambridge LCF, Technical Report 39, University of Cambridge, Computer Laboratory.

Paulson L. C. [1986], 'Natural deduction as higher-order resolution', *Journal of Logic Programming* **3**, 237–258.

Paulson L. C. [1989], 'The foundation of a generic theorem prover', *Journal of Automated Reasoning* **5**(3), 363–397.

Paulson L. C. [1990], Isabelle: The next 700 theorem provers, *in* P. Odifreddi, ed., 'Logic and Computer Science', Academic Press, pp. 361–386.

Paulson L. C. [1993], Isabelle's object-logics, Technical Report 286, University of Cambridge, Computer Laboratory.

Paulson L. C. [1994], *Isabelle: A Generic Theorem Prover*, Springer-Verlag LNCS 828.

Petersson K. [1982], A programming system for type theory, PMG Report 9, Chalmers University of Technology.

Pfenning F. [1989], Elf: A language for logic definition and verified meta-programming, *in* 'Fourth Annual Symposium on Logic in Computer Science', IEEE Computer Society Press, Pacific Grove, California, pp. 313–322.

Pfenning F. [1991*a*], Logic programming in the LF logical framework, *in* G. Huet and G. Plotkin, eds, 'Logical Frameworks', Cambridge University Press, pp. 149–181.

Pfenning F. [1991*b*], Unification and anti-unification in the Calculus of Constructions, *in* 'Sixth Annual IEEE Symposium on Logic in Computer Science', Amsterdam, The Netherlands, pp. 74–85.

Pfenning F. [1993], Refinement types for logical frameworks, *in* H. Geuvers, ed., 'Informal Proceedings of the Workshop on Types for Proofs and Programs', Nijmegen, The Netherlands, pp. 285–299.

Pfenning F. [1994*a*], Elf: A meta-language for deductive systems, *in* A. Bundy, ed., 'Proceedings of the 12th International Conference on Automated Deduction', Springer-Verlag LNAI 814, Nancy, France, pp. 811–815. System abstract.

Pfenning F. [1994*b*], Structural cut elimination in linear logic, Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University.

Pfenning F. [1995], Structural cut elimination, *in* D. Kozen, ed., 'Proceedings of the Tenth Annual Symposium on Logic in Computer Science', IEEE Computer Society Press, San Diego, California, pp. 156–166.

Pfenning F. [1996], The practice of logical frameworks, *in* H. Kirchner, ed., 'Proceedings of the Colloquium on Trees in Algebra and Programming', Springer-Verlag LNCS 1059, Linköping, Sweden, pp. 119–134. Invited talk.

Pfenning F. [2000], 'Structural cut elimination I. Intuitionistic and classical logic', *Information and Computation* **157**(1/2), 84–141.

Pfenning F. [2001], *Computation and Deduction*, Cambridge University Press. In preparation. Draft from April 1997 available electronically.
URL: *http://www.cs.cmu.edu/~twelf/notes/cd.ps*

Pfenning F. and Elliott C. [1988], Higher-order abstract syntax, *in* 'Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation', Atlanta, Georgia, pp. 199–208.

Pfenning F. and Rohwedder E. [1992], Implementing the meta-theory of deductive systems, *in* D. Kapur, ed., 'Proceedings of the 11th International Conference on Automated Deduction', Springer-Verlag LNAI 607, Saratoga Springs, New York, pp. 537–551.

Pfenning F. and Schürmann C. [1998*a*], Algorithms for equality and unification in the presence of notational definitions, *in* T. Altenkirch, W. Naraschewski and B. Reus, eds, 'Types for Proofs and Programs', Springer-Verlag LNCS 1657, Kloster Irsee, Germany, pp. 179–193.

Pfenning F. and Schürmann C. [1998*b*], 'Twelf', Project home page. Version 1.2.
URL: *http://www.cs.cmu.edu/~twelf*

Pfenning F. and Schürmann C. [1998*c*], *Twelf User's Guide*, 1.2 edn. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.

Pinto L. and Dyckhoff R. [1998], Sequent calculi for the normal terms of the $\lambda\pi$- and $\lambda\pi\sigma$-calculi, *in* D. Galmiche, ed., 'Proceedings of the Workshop on Proof Search in Type-Theoretic Languages', Vol. 17 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, Lindau, Germany.
URL: *http://www.elsevier.com/locate/entcs/volume17.html*

Pollack R. [1994], The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions, PhD thesis, University of Edinburgh.

λProlog [1997], Home page. Indexes lambda Prolog implementations.
URL: *http://www.cse.psu.edu/~dale/lProlog/*

Pym D. [1990], Proofs, Search and Computation in General Logic, PhD thesis, University of Edinburgh. Available as CST-69-90, also published as ECS-LFCS-90-125.

Pym D. [1992], 'A unification algorithm for the $\lambda\Pi$-calculus', *International Journal of Foundations of Computer Science* **3**(3), 333–378.

Pym D. and Wallen L. [1990], Investigations into proof-search in a system of first-order dependent function types, *in* M. Stickel, ed., 'Proceedings of the 10th International Conference on Automated Deduction', Springer-Verlag LNCS 449, Kaiserslautern, Germany, pp. 236–250.

Pym D. and Wallen L. A. [1991], Proof search in the $\lambda\Pi$-calculus, *in* G. Huet and G. Plotkin, eds, 'Logical Frameworks', Cambridge University Press, pp. 309–340.

Qian Z. [1993], Linear unification of higher-order patterns, *in* M.-C. Gaudel and J.-P. Jouannaud, eds, 'Proceedings of the Colloquium on Trees in Algebra and Programming', Springer-Verlag LNCS 668, Orsay, France, pp. 391–405.

Rohwedder E. and Pfenning F. [1996], Mode and termination checking for higher-order logic programs, *in* H. R. Nielson, ed., 'Proceedings of the European Symposium on Programming', Springer-Verlag LNCS 1058, Linköping, Sweden, pp. 296–310.

Ruess H. [1996], Reflection of formal tactics in a deductive reflection framework, *in* M. McRobbie and J. Slaney, eds, 'Proceedings of the 13th International Conference on Automated Deduction', Springer-Verlag LNAI 1104, New Brunswick, New Jersey, pp. 628–642.

Ruess H. [1997], Computational reflection in the calculus of constructions and its application to theorem proving, *in* P. de Groote and R. Hindley, eds, 'Proceedings fo the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)', Springer-Verlag LNCS, Nancy, France, pp. 319–335.

Schroeder-Heister P. [1991], Structural frameworks, substructural logics, and the role of elimination inferences, *in* G. Huet and G. Plotkin, eds, 'Logical Frameworks', Cambridge University Press, pp. 385–403.

Schroeder-Heister P. [1993], Rules of definitional reflection, *in* M. Vardi, ed., 'Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science', Montreal, Canada, pp. 222–232.

Schürmann C. [1995], A computational meta logic for the Horn fragment of LF, Master's thesis, Carnegie Mellon University. Available as Technical Report CMU-CS-95-218.

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

Schürmann C. and Pfenning F. [1998], Automated theorem proving in a simple meta-logic for LF, *in* C. Kirchner and H. Kirchner, eds, 'Proceedings of the 15th International Conference on Automated Deduction (CADE-15)', Springer-Verlag LNCS 1421, Lindau, Germany, pp. 286–300.

Shankar N. [1988], 'A mechanical proof of the Church-Rosser theorem', *Journal of the Association for Computing Machinery* **35**(3), 475–522.

Troelstra A. S. and van Dalen D. [1988], *Constructivism in Mathematics*, Vol. 121 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, Amsterdam.

Virga R. [1996], Higher-order superposition for dependent types, *in* H. Ganzinger, ed., 'Proceedings of the 7th International Conference on Rewriting Techniques and Applications', Springer-Verlag LNCS 1103, New Brunswick, New Jersey, pp. 123–137. Extended version available as Technical Report CMU-CS-95-150, May 1995.

Virga R. [1999], Higher-Order Rewriting with Dependent Types, PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University. Available as Technical Report CMU-CS-99-167.

Index