2000

Logical Frameworks: Why Not Just Classical Logic?

Iliano Cervesato
Carnegie Mellon University

Logical Frameworks

Why not just classical logic?

Iliano Cervesato

I was recently invited to give a presentation about the logical framework *LLF*. After a 40 minutes talk in which I revealed the intricacies of the underlying type theory and illustrated by means of examples the meta-representation wonders of this new language, somebody in the audience said: "This looks very complicated. Why not using, say, classical logic instead?". In this chapter, I build upon my then improvised answer. I will recall what logical frameworks are and try to motivate the simple but unfamiliar constructs they often rely upon.

1 Introduction

It is often taught in introductory classes in Philosophy, Mathematics, and Computer Science that logic is the universal language of reasoning and rigorous representation of knowledge. This is not unfounded: for example, the entire body of mathematics can be formalized in classical first-order predicate logic.

This however causes people with only a basic logical background to frown when they hear logical framework experts, those scholars who specialize in formal reasoning and knowledge representation, use scary-sounding words such as "higher-order abstract syntax", "dependent type theories", and "linearity". If classical logic is so universal, why do these authors rebuff it for those apparently cryptic and hopelessly complicated languages?

In this chapter, we show that many ideas in modern logical frameworks emerge as refinements of computationally or representationally suboptimal aspects of classical logic. Our progression does not reflect the historical development of these sys-

Proceedings of the Seventh CSLI Workshop on Logic, Language and Computation.
Martina Faller, Stefan Kaufmann & Marc Pauly (eds.)

tems, and is geared towards logical frameworks in the *LF* family [Harper 1993, Pfenning 1991].

2 Deductive Systems

Exchanging information and reasoning about a formalism, may it be a novel programming language or an arcane algebraic structure, presupposes that we have a language to express the concepts we are interested in. These include basic linguistic facts such as "$3 + 5$ is a well-formed expression" (*syntax*), descriptions of the effect of operations, for example "$3 + 5$ evaluates to 8" (*semantics*), and properties of the formalism such as "$+$ is commutative" (*meta-theory*). In these three examples, we used English to express the concepts involved. Being fully rigorous when using natural languages is difficult to achieve and impossible to enforce. Jargon and abbreviations (e.g. writing "$3 + 5 \hookrightarrow 8$" for "$3 + 5$ evaluates to 8") only alleviate the problem.

Deductive systems [Martin-Löf 1985] approach rigor by fixing precise conventions on how to express the concepts of the formalism we are interested in (the *object language*). In those presentations, the predications we make about object entities are called *judgments*. The quoted examples above are all judgments.

In their simplest instance, judgments are syntactic descriptions of relations among lexical elements of the object language. They can either *hold* (e.g. "$3 + 5 \hookrightarrow 8$") or *fail to hold* (e.g. "$3 + 5 \hookrightarrow 9$"). Expressing the *meaning* of the constructs of the object language amounts then to specifying which of the relevant judgments hold, and which do not. An explicit and exhaustive enumeration of the former (and of the latter) is generally ineffective since infinitely many judgments may be involved ($\_ + \_ \hookrightarrow \_$ is an example). However, the constructs commonly found in all formalisms of interest display forms of regularity that make them amenable to a finite description. Schematic *rules of inference* achieve this effect by expressing the validity of classes of judgments sharing a common syntactic pattern (*rule conclusion*) in terms of the validity of zero or more other judgments (*rule premises*). Instances of inference rules are chained to provide ev-

idence of the validity of specific judgments in the form of finite *derivations*. A judgment is then *derivable* if it has a derivation. A deductive system faithfully expresses an object formalism if all and only the judgments that hold are derivable. We will now illustrate these definitions on our running example.

Notations for grammars such as the Backus-Naur Form yield deductive systems for the syntax of a formalism. If, for the sake of succinctness, we express natural numbers in unary notation, we can specify the syntax of our example as follows:

$$\begin{array}{l} \mathit{num} ::= \mathrm{z} \mid \mathrm{s}\ \mathit{num} \\ \mathit{exp} ::= \mathit{num} + \mathit{num} \end{array}$$

where we have chosen the symbols z, s, and + to denote zero, successor, and addition, respectively. The implicit judgments here are "the string _ is a number" and similarly for expressions. The inference rules are the grammatical productions: for example, the first line states that "z is a number" (rule with no premises), and that "s N is a number" if "N is a number" (rule schematic in N with one premise). The derivations are all the parse trees.

Here, "s" can be viewed either as a character (or token) so that "s s z" is a list of characters (tokens), or as a function symbol with one arguments so that "s s z" stands for the expression "s (s (z))". The former approach is called *concrete syntax*, the latter *abstract syntax*. Expressions in the abstract syntax are isomorphic to parse-trees.

Given this representation of numbers, a simple recursive definition specifies how to evaluate their sum: "adding zero to any number N yields N", and "for any numbers M, N, and V, s $M + N$ evaluates to s V if $M + N$ evaluates to V". Transliterating each part into symbols yields the following two rules:

$$\frac{N\ \mathit{num}}{\mathrm{z} + N \hookrightarrow N}\ \textbf{eval\_z} \qquad \frac{N, M, V\ \mathit{num} \qquad M + N \hookrightarrow V}{\mathrm{s}\ M + N \hookrightarrow \mathrm{s}\ V}\ \textbf{eval\_s}$$

The horizontal line separates the premises and the conclusion of each rule, and the text on the right identifies the rule. The first rule has one premise and is schematic in N, the second has four (abbreviated) premises and is schematic in M, N, and V. It

is easy to observe that the syntactic judgments in rule **eval_s** are redundant, but omitting the premise of rule **eval_z** would allow successfully evaluating garbled expressions (e.g. $z + oops \hookrightarrow oops$). Observe that if we think of each rule as describing an atomic step of the evaluation of the sum of two numbers, then derivations are a notation for evaluation traces.

Meta-theoretic properties are predications over semantic derivations. For example, "+ is commutative" can be restated as "given numbers M, N, and V, for every derivation $\mathcal{D}$ of $M+N \hookrightarrow V$, there exists a derivation $\mathcal{D}'$ of $N + M \hookrightarrow V$". Therefore, a convenient notation for derivations is an essential prerequisite for reasoning about a formalism. For space reasons, we refrain from further discussing meta-theoretic judgments (properties) and their derivations (proofs). The techniques we will illustrate are however applicable also in that setting (see [Michaylov 1991]).

3 Logical Frameworks

Through the notions of judgment and derivation, deductive systems allow precise descriptions of formalisms in Mathematics, Logic, and Computer Science. However, when exchanging ideas with others or proving properties, we seldom adhere to their full formality: their rigid patterns soon get in the way of effective communication. A variously balanced mixture of natural language, judgments, and derivation sketches is normally adopted as a good compromise between rigor and bearability.

Formalizing even simple proofs often requires a fair amount of work with little benefit: indeed, the formal argument is seldom more convincing than the original proof since we, as humans, have a limited ability of keeping alert when confronted with long and convoluted chains of inference. Paradoxically, formal errors are more likely to pass unnoticed than informal ones.

Since computers are free from the attention shortcomings of the human brain, they are ideal candidates for the clerical work of checking proofs and derivations, and, in simple cases, of validating judgments. Parsers, interpreters, compilers, and various related tools efficiently mechanizes aspects of the syntax, seman-

tics, and meta-theory of specific languages. The research on logical frameworks explores instead ways of automating common issues found in generic deductive systems. Not surprisingly, major developments were made in the area of general purpose theorem proving.

Logical frameworks are formalisms specifically tailored for representing and reasoning about generic deductive systems in an automated environment. Their chief constituent is a *meta-language*, a deductive systems itself, that serves the purpose of encoding the judgments and derivations of the object languages. They also come with a body of techniques, known as their *representation methodology*, that dictates how to best express the different aspects of an object deductive system.

Not every formalism capable of representing generic formal knowledge is suitable as the basis for a logical framework. A good meta-language must satisfy two additional requirements: *executability*, i.e. it should have good computational properties (this excludes for example general set theory or specification languages such as *Z*), and *immediacy*, i.e. it should provide simple and transparent support for representing the basic constitutents of a deductive system, judgments and derivations (this excludes not only approaches based on Gödel numbering and Turing machines, but also many common programming languages such as *C* or *Java*). We will now make these requisites more precise.

The minimal computational requirement of a meta-language is that checking whether an expression is the representation of a well-formed derivation for a given judgment should be decidable. Since proofs are meta-theoretic derivations, this also means that the system should be able to decide whether an (encoded) proof is actually correct. This is the extent of what *AUTOMATH* [deBrujn 1980], one of the first logical frameworks, was able to do. More recent systems also mechanize aspects of the discovery of a derivation for a given judgment, an activity that includes proof-search. These frameworks fall roughly into two categories: some are designed as interactive theorem provers (e.g. *NuPrl* [Constable 1986], *Coq* [Dowek 1993], and *Isabelle* [Paulson 1993]), while others are implemented as logic pro-

gramming languages (e.g. λ*Prolog* [Miller] and *Twelf* [Twelf]).

Opting for a representation that closely resembles the object system it encodes has several advantages. First and most importantly, it makes it easy to check that the representation faithfully models the different constructs of the object language and their behavior. Many authors actually provide *adequacy theorems* and their proofs as evidence of the correctness of their representation. Second, it yields simple, short, elegant, and fast formalizations of the object language (a deductive system that takes mere hours to encode in one meta-language can require months in another). Third, when doing proof-search, a more complex encoding generally entails a larger search space, and therefore exponentially more time can be needed in order to find a solution.

A deductive system can be given an immediate representation in a logical framework if the meta-language provides primitive operations for all the forms of judgment and derivation it mentions (observe that there exist more complex judgments than the ones we saw in Section 2). In this way, the encoding can focus on the object language rather than on the infrastructure that expresses it. As of now, there is no meta-language that embeds satisfactorily all the patterns that can appear in a deductive system. Instead, current logical frameworks provide internal support for different features, which makes them adequate for different classes of object formalisms. In the sequel, we will examine some of the most recurrent features and present a succession of meta-languages that incorporate them. Classical logic (actually a sublanguage of it) will be our starting point.

4 Horn Clauses

First-order classical logic has been used for decades to represent and reason about formal systems in various domains. Moreover, although proving the derivability of a formula is undecidable, there exist fairly sofisticated theorem provers that have been successfully employed for this task. In the light of these facts, giving a simple and faithful representation of deductive systems like the one in Section 2 seems trivially within the reach of this formalism.

Let us try. Natural numbers and expressions can be represented by encoding either their concrete or abstract syntax. In the former case, we use the binary function $\mathsf{cons}(\_,\_)$ and the constant nil to construct lists of symbols, emulating strings. We furthermore choose a constant for every token, here z, s, and $+$ for "z", "s", and "+", respectively. "s s z" is then represented as $\mathsf{cons}(\mathsf{s}, \mathsf{cons}(\mathsf{s}, \mathsf{cons}(\mathsf{z}, \mathsf{nil})))$.

The abstract syntax yields a more direct encoding. In this case, we represent "z", "s", and "+" as the constant z, the unary function symbol $\mathsf{s}(\_)$, and the binary function symbol $+(\_,\_)$, respectively. Now, the expression "s s z" is entered as $\mathsf{s}(\mathsf{s}(\mathsf{z}))$.

Notice that, differently from the previous representation, we have no way to express garbled expressions such as "z s +". This is however acceptable if we are only interested in well-formed entities. Observe also that this notation provides a direct access to the subexpressions of the operators: for example, if we want to access the second operand of "+" in the concrete representation of the expression "$N + M$", for some M and N, we need to explicitly provide a procedure that strips the unwanted prefix; on the other hand, it suffices to select the second argument of the $+(\_,\_)$ function symbol in the abstract representation. Considering how frequently inference rules refer to logical subterms of their constitutents, we are almost compelled to adopt encodings that mimick the abstract syntax. This is the first situation in which immediacy guides our representational choices.

The three judgments appearing in our example, "$\_$ *num*", "$\_$ *exp*", and "$\_ \hookrightarrow \_$", can be encoded as the three predicate symbols $\mathsf{num}(\_)$, $\mathsf{exp}(\_)$, and $\mathsf{eval}(\_,\_)$, respectively. Then, the judgment "s s z + s z $\hookrightarrow$ s s s z" is mapped to the predicate $\mathsf{eval}(+(\mathsf{s}(\mathsf{s}(\mathsf{z})), \mathsf{s}(\mathsf{z})), \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))))$.

The inference rules appearing in our example have a very simple form: they all express the fact that, for any instantiation of the schematic variables, if all the premises are derivable, then there is a derivation of the conclusion. It is then natural to encode them by means of formulas of the form

$$\forall x_1 \ldots x_n. A_1 \wedge \ldots \wedge A_m \Rightarrow B,$$

where B is the representation of the rule conclusion, $A_1, \ldots, A_m$ encode its premises, and $x_1, \ldots, x_n$ are all its schematic variables. Rules with no premises are rendered as the abbreviated $\forall x_1 \ldots x_n.\, B$. Formulas of this form are known as *Horn clauses*. There exist efficient proof-search algorithms for them, to the extent that they constitute the declarative core of the logic programming language *Prolog* [Sterling 1994].

The inference figures appearing in our example are encoded as the following Horn clauses:

$$\mathsf{num}(\mathsf{z}).$$
$$\forall N.\, \mathsf{num}(N) \Rightarrow \mathsf{num}(\mathsf{s}\,(N)).$$
$$\forall M, N.\, \mathsf{num}(M) \wedge \mathsf{num}(N) \Rightarrow \mathsf{exp}(+(M, N)).$$
$$\forall N.\, \mathsf{num}(N) \Rightarrow \mathsf{eval}(+(\mathsf{z}, N), N).$$
$$\forall M, N, V.\, \mathsf{num}(M) \wedge \mathsf{num}(N) \wedge \mathsf{num}(V) \wedge \mathsf{eval}(+(M, N), V) \Rightarrow \mathsf{eval}(+(\mathsf{s}\,(M), N), \mathsf{s}\,(V)).$$

A *Prolog* system can check whether a predicate follows logically from these formulas. By virtue of adequacy theorems that space limitation prevent us from stating, a judgment from our example is derivable if and only if the corresponding predicate is a logical consequence of the above Horn clauses.

This provides a way of verifying derivability, but we did not mention derivations yet. We can however painlessly alter the above representation to validate them. In order to do so, we associate to every inference rule a unique function symbol with as many arguments as the number of its schematic variables and premises. In the case of rule **eval_z**, we will have for example $\mathsf{eval\_z}(\_, \_)$. The presence of this constant in the representation of a derivation acts as a witness of the application of that rule. Its arguments are intended to hold the instantiation of the schematic variables and the representation of the derivation of each premise. For example, a derivation of the judgment "$\mathrm{s\,z} + \mathrm{s\,z} \hookrightarrow \mathrm{s\,s\,z}$" is represented as the following term:

$$\begin{array}{l}\mathsf{eval\_s}\,(\mathsf{z},\ \mathsf{s}\,(\mathsf{z}),\ \mathsf{s}\,(\mathsf{z}),\\ \qquad \mathsf{num\_z},\ \mathsf{num\_s}\,(\mathsf{z}, \mathsf{num\_z}),\ \mathsf{num\_s}\,(\mathsf{z}, \mathsf{num\_z}),\\ \qquad \mathsf{eval\_z}\,(\mathsf{s}\,(\mathsf{z}),\\ \qquad\qquad \mathsf{num\_s}\,(\mathsf{z}, \mathsf{num\_z}))\end{array}$$

The derivation starts with rule **eval_s** as indicated by the use of the symbol eval_s. Its first three arguments (first line) correspond to the instantiations of the schematic variables M, N and V of that rule. The next three arguments (second line) encode derivations of the syntactic premises we abbreviated as $N, M, V\ num$. The last argument (third and fourth line) represents a derivation of the premise $M + N \hookrightarrow V$, which consists of an application of rule **eval_z**.

In order to verify the validity of a derivation, we include it as an additional argument to the representation of every judgment. Then, we have the clause encoding each rule check that the corresponding function symbol appears in the derivation. For example, in the case of **eval_z**, we obtain:

$$\forall N, D.\, \mathsf{num}(N, D) \Rightarrow \mathsf{eval}(+(\mathsf{z}, N), N, \mathsf{eval\_z}(N, D)).$$

The correctness of an evaluation derivation can now be verified by encoding it in the extra argument of the predicate eval and having *Prolog* verify whether it is a logical consequence of the enriched clauses. Derivability can still be established by proving formulas of the form $\exists D.\, \mathsf{eval}(+(M, N), V, D)$ for given M, N, and V. *Prolog* can do even better and construct such a D (as well as the value V if left uninstantiated).

5 Sorts

An attentive observation of the clauses for eval in the previous section reveals that the lexical predicates appearing in them serve the sole purpose of ascertaining that the instantiation of their variables belong to the proper syntactic category. Removing them would cripple the representation by validating non-derivable judgments. However, evaluation is intended only to relate expressions to natural numbers. Therefore, we could get rid of these predicates if the first argument of eval were constrained to be an expression, and its second argument a number.

The syntactic clauses themselves are unsatisfactory: the second formula for example is intended to express the fact that the function symbols s accepts a natural number as an argument

to produce a natural number; at the same time, the ill-formed expression $\mathsf{s}\,(+(\mathsf{z},\mathsf{z}))$ is legitimate, although non-derivable.

In both cases, the cause of our discontent is that the notion of term available in classical first-order logic is too coarse to syntactically prevent the formation of meaningless expressions. Therefore, since the arguments of a function or predicate symbol can be any term, we must rely on dedicated formulas to weed out unwanted expressions.

We can solve these problems by moving to a *multi-sorted* version of the language of Horn clauses. This formalism makes it possible to declare *sorts* for the purpose of classifying terms. Every constant should then be assigned a sort, we should state the sorts of the arguments and of the result of function symbols, and similarly we should give the sort of the arguments of predicate symbols. The declared sorts and the resulting *type* of each symbol are collected in a *signature*. The clauses change only to the extent that we must specify the sort of every bound variable.

In the case of our example, we obtain the following formalization, where the keyword type is used to declare sorts, and the rest of the notation should be self-explanatory:

$$\left.\begin{array}{l}\mathsf{num} : \mathsf{type}.\\ \mathsf{exp} : \mathsf{type}.\\ \mathsf{z} : \mathsf{num}.\\ \mathsf{s} : \mathsf{num} \to \mathsf{num}.\\ + : \mathsf{num} \times \mathsf{num} \to \mathsf{exp}.\\ \mathsf{eval} : \mathsf{exp} \times \mathsf{num}.\end{array}\right\}\ \text{Signature}$$

$$\forall N{:}\mathsf{num}.\, \mathsf{eval}(+(\mathsf{z}, N), N).$$

$$\forall M, N, V{:}\mathsf{num}.\, \mathsf{eval}(+(M, N), V) \Rightarrow \mathsf{eval}(+(\mathsf{s}\,(M), N), \mathsf{s}\,(V)).$$

Now, the expression $\mathsf{s}\,(+(\mathsf{z},\mathsf{z}))$ is rejected since s expects an argument of sort num, while $+(\mathsf{z},\mathsf{z})$ has type exp. Observe that syntactic judgments are not represented any more by predicates but by types. It can be proved that every natural number is encoded as a unique term of sort num, and that every term of this type is the representation of some natural number. A similar statement holds for expressions. This representation is computationally adequate since type checking is decidable in this language.

The syntactic category of the variables in the two clauses

remaining is now constrained up-front rather than by means of a formula. Observe however that the type of these variables could be omitted since it is determined by the context in which they occur. This is a general property of multi-sorted first-order logic.

Evaluation derivations are represented in this setting as in the previous section. The only difference is that we need to declare a sort for them (deriv, say) and instrument the required constants with the proper types.

6 Higher-Order Terms

The representation we obtained so far is admittedly elegant, but the object language itself is not very interesting. We can make it more useful by introducing other arithmetic operators or by including conditionals and a supply of boolean constructs. The techniques we examined so far apply smoothly in all these cases.

Adding constructs that bind variables, such as functions, quantifiers, or simply local variables, is not as painless however. In this section, we will consider an object language consisting of the latter, plus any of the construct we mentioned above. We have the following grammar (notice that we are doing without the lexical category *num*):

$$exp ::= \mathrm{z} \mid \mathrm{s}\ exp \mid \ldots \mid x \mid \mathrm{let}\ x = exp\ \mathrm{in}\ exp$$

here, x ranges over the new lexical category of *variables*. The expression "let $x = E_1$ in E_2" declares x as a local variable of E_2 and initializes it to the value of E_1. Locality means that x is not visible outside E_2 even when the "let" expression is part of a larger term. If two "let" constructs with the same local identifier are nested, the inner one shadows the outer one within its scope.

Evaluating the above construct amounts to computing the value of E_1, substituting it for x in E_2, and then evaluating the resulting expression. We have the following inference rule:

$$\frac{E_1 \hookrightarrow V' \qquad [V'/x]E_2 \hookrightarrow V}{\mathrm{let}\ x = E_1\ \mathrm{in}\ E_2 \hookrightarrow V}\ \textbf{eval\_let}$$

where $[V'/x]E_2$ is our informal notation for the substitution of every free occurrence of x in E_2 with V'. "let"s are a convenient way to factor out large or recurrent subexpressions.

A first attempt at encoding the "let" construct in multi-sorted first-order logic may add the following declarations to the encoding in Section 4:

$$
\begin{array}{l}
\mathsf{var} : \mathsf{type}. \\
\mathsf{var2exp} : \mathsf{var} \to \mathsf{exp}. \\
\mathsf{let} : \mathsf{var} \times \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}. \\
\mathsf{subst} : \mathsf{exp} \times \mathsf{var} \times \mathsf{exp} \times \mathsf{exp}. \\
\\
\forall X{:}\mathsf{var}.\, \forall E_1, E_2, E_2', V, V'{:}\mathsf{exp}. \\
\quad \mathsf{eval}(E_1, V') \wedge \mathsf{subst}(V', X, E_2, E_2') \wedge \mathsf{eval}(E_2', V) \\
\qquad \Rightarrow \mathsf{eval}(\mathsf{let}(X, E_1, E_2), V).
\end{array}
$$

where var is the sort of variables, var2exp casts a variable into an expression, and "let $x = e_1$ in e_2" is represented as the term $\mathsf{let}(X, E_1, E_2)$, where X, E_1, and E_2 are the encodings of x, e_1, and e_2, respectively. The clause for eval represents rule **eval_let**. Notice that it relies on the auxiliary predicate subst to realize substitution. For the sake of conciseness, we did not show the clauses for subst.

Although this encoding represents our intuition about "let", it has several inconvenients. First, nothing prevents a variable from escaping its scope: for example, assuming that x has been declared as a variable, $+(\mathsf{x}, \mathsf{let}(\mathsf{x}, \mathsf{z}, \mathsf{s}\,(\mathsf{x})))$ is a legitimate term although it is ill-formed if the variable x is not intended to appear outside of the "let" expression. We can track this problem to the fact that our representation schema forces us to declare all identifiers in the global signature: we have no way to specify that x should exist only inside the third argument of the let.

A second source of unsatisfaction derives from the necessity of axiomatizing the definition of substitution. Had we displayed them, the clauses for subst would have amounted to more than half of the encoding of the entire example. Their specification is tricky since they must go inside terms, properly handle homonymous variables, and in situations slightly more complicated than ours perform variable renaming in order to avoid capture. Derivation encodings are proportionally more complicated, causing rea-

soning correctly about them to often be overwhelming. This goes against the ideal of immediacy that we are promoting, in particular considering how frequently binding constructs appear in deductive systems.

These considerations lead us to look for a meta-language that offers a primitive support for the notions of local variable, binder, and substitution. This is achieved by replacing the (multi-sorted) first-order term language we have been using with a *simply-typed* λ*-calculus*.

Recall that a term $\lambda X{:}\,\tau.\,M$ binds the variable X (of type τ) in the sub-expression M: X is local to M and undefined elsewhere. The idea is then to adopt the λ-abstraction operator to encode the binding effect of generic object-language constructs that introduce local variables. For example, we will represent the expression "let $x = \mathsf{z}$ in $\mathsf{s}\,x$" as the term $\mathsf{let}(\mathsf{z}, \lambda X{:}\mathsf{exp}.\,\mathsf{s}\,(X))$. Notice that the object-level variable x is simply represented as a variable X of the meta-language, and that X is given the correct type exp. In order to achieve this representation, we must endow the second argument of let with a functional type. This symbol is indeed declared of type $\mathsf{exp} \times (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$.

This addresses the first issue above. Our frustration with substitution has a simple solution as well since, in the λ-calculus, the terms $(\lambda X{:}\tau.\,M)N$ and $[N/X]M$ are considered equal for generic expressions M and N, and variable X (they are β*-equivalent*). Therefore, whenever an object-level inference rule performs a substitution, we can emulate it as a meta-language application, a primitive operation of the λ-calculus.

The presence of λ-abstraction allows typable terms that do not correspond to any object language expression. A simple example is $(\lambda X{:}\mathsf{exp}.\,X)\ \mathsf{z}$, which has type exp. However, it is possible to prove that each such term is equivalent to a unique *canonical form* (z in this case) representing some object-level expression (here z). Indeed, the adequacy theorem for our example states that every expression is encoded as a unique canonical term of sort exp and that every such term is the representation of a distinct expression.

The representation methodology we just illustrated is called

higher-order abstract syntax. This technique yields the following elegant encoding in the case of our example:

$$\begin{array}{l}\mathsf{let} : \mathsf{exp} \times (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}. \\ \forall E_1, V, V' : \mathsf{exp}.\, \forall E_2 : \mathsf{exp} \to \mathsf{exp}. \\ \quad \mathsf{eval}(E_1, V') \wedge \mathsf{eval}(E_2'\, V', V) \Rightarrow \mathsf{eval}(\mathsf{let}(E_1, E_2), V).\end{array}$$

Recall here that E_2 is a term of the form $\lambda X : \mathsf{exp}.\, E_2'$. Therefore, the application $E_2'\, V'$ reduces to the desired substitution $[V'/X]E_2'$. An encoding of derivations can be added to this representation as in the previous sections.

7 Hereditary Harrop Formulas

As our object language is growing, we may be interested in distinguishing between expressions standing for natural numbers and expressions denoting booleans, for example. Hardwiring this classification in the grammar of the object formalism is unsatisfactory because of constructs such as "let" that operate on generic expressions. A more flexible approach is to introduce a notion of *type* in the object language. For example, our types can be:

$$T ::= \mathrm{nat} \mid \mathrm{bool}$$

We then extend the semantics of the object language with a new *typing judgment* that associates a type to each expression that is to be considered legal. We denote it as $\Gamma \vdash E : T$, where E is an object-level expression, T is its type, and Γ is a *context* that gives the type of every free variables in E. Here are some of the rules that define the derivability of this judgment:

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \textbf{hasType\_var} \qquad \frac{}{\Gamma \vdash \mathrm{z} : \mathrm{nat}}\ \textbf{hasType\_z}$$

$$\frac{\Gamma \vdash E_1 : T' \qquad \Gamma, x : T' \vdash E_2 : T}{\Gamma \vdash \mathrm{let}\ x = E_1\ \mathrm{in}\ E_2 : T}\ \textbf{hasType\_let}$$

Notice the way the context is used to assess the type of a variable, and how (variable,type) pairs enter it when analyzing a binding construct.

We want to formalize this judgment and its derivations in our meta-language. The novel aspect here is the context. A first idea is to represent it as a list of (variable,type) pairs. This does not work because the language introduced in the previous section does not provide ways to declare free variables; using λ-bound variables is not very promising either since we would need means to operate within arbitrarily many nested abstractions. A second idea is to use some kind of index, progressive numbers for example: in the case of the right premise of rule **hasType_let**, we would pick a new index i, store it together with T' in the representation of the context, substitute it for x in E_2, and proceed with this new expression. A specification of rule **hasType_var** would dismantle the representation of the context to find the right (index,type) pair.

Although correct, this encoding is not very immediate. A closer look at rule **hasType_let** suggests a better representation: its right premise can be read as stating that for an arbitrary expression (indicated as x), if that expression has type T' (the new context item $x : T'$), then E_2 (which may mention x) has type T. Under this light, this judgment is *parametric* in x (and its other free variables) and *hypothetical* in $x : T'$ (and in the other pairs in Γ).

This reading can easily be embedded in the meta-language we have been using by being more liberal about the encoding of the premises of a rule. Rather than just a predicate A, we will now allow a premise to be represented by a formula of the following form:

$$\forall y_1{:}\tau_1, \ldots, y_p{:}\tau_p.\, C_1 \Rightarrow \ldots \Rightarrow C_q \Rightarrow A.$$

The variables $y_1, \ldots, y_p$ will stand for the parameters introduced in the premise, with their sorts, and $C_1, \ldots, C_q$ for its new assumptions. If $p = 0$ and $q = 0$, this formula reduces to A, our previous encoding of plain judgments. This extension to the representation of inference rules takes us outside of the language of Horn clauses, but still within the borders of the the language of *hereditary Harrop formulas* [Miller 1991], another formalism with excellent computational properties. It has been implemented as the logic programming language λ*Prolog* [Miller]. As a technical

note, our new meta-language (even without sorts and higher-order terms) is a fragment of intuitionistic logic, but not of classical logic; instead, Horn clauses belong to both.

The application of this technique to the rules above yields the following specification:

$$\begin{array}{l}
\mathsf{tp} : \mathsf{type}. \\
\mathsf{nat}, \mathsf{bool} : \mathsf{tp}. \\
\mathsf{hasType} : \mathsf{exp} \times \mathsf{tp}. \\
\\
\mathsf{hasType}(\mathsf{z}, \mathsf{nat}). \\
\forall E_1{:}\mathsf{exp}.\, \forall E_2{:}\mathsf{exp} \to \mathsf{exp}.\, \forall T, T'{:}\mathsf{tp}. \\
\quad \mathsf{hasType}(E_1, T') \wedge (\forall X{:}\mathsf{exp}.\, \mathsf{hasType}(X, T') \Rightarrow \mathsf{hasType}(E_2\ X, T)) \\
\qquad \Rightarrow \mathsf{hasType}(\mathsf{let}(E_1, E_2), T).
\end{array}$$

Proving the formula $\forall X{:}\mathsf{exp}.\, \mathsf{hasType}(X, T') \Rightarrow \mathsf{hasType}(E_2 X, T)$, which encodes the second premise of rule **hasType_let**, can be seen as temporarily augmenting the signature with the declaration $\mathsf{x}{:}\,\mathsf{exp}$ for some new constant x, enriching the set of clauses with $\mathsf{hasType}(\mathsf{x}, T')$ (as if it were the representation of a new rule), and then finding a proof of $\mathsf{hasType}(E_2\mathsf{x}, T)$ where the bound variable of E_2 is substituted with x. Notice that, coherently with this interpretation, there is no representation for rule **hasType_var**: it is realized by having assumed a dedicated typing assumption for each introduced parameter.

Having changed the nature of the formulas that encode inference rules, we need to revise our representation of derivations. In order to do so, consider a parametric and hypothetical judgment of the form $\Gamma \vdash E : T$. A free variable x in it can be seen as placeholders for an actual expression E'; similarly, the corresponding pair $x : T'$ in Γ can be viewed as a placeholder for an actual derivation $\mathcal{D}'$ of the fact that E' has type T'. Therefore, a derivation $\mathcal{D}$ of $\Gamma \vdash E : T$ differs from the derivations we have encountered so far by the fact that it is abstract with respect to the instantiation of its parameters and the derivations of its assumptions. This can be emulated by using the λ-abstraction operator of our term language (we omit details for space reasons). For example, a derivation of the judgment

$$\cdot \vdash \text{let } x = \mathsf{z} \text{ in } \mathsf{s}\,(x) : \text{nat}$$

(where $\cdot$ denotes the empty context) is represented by the following term:

$$\begin{array}{l}\mathsf{hasType\_let}\,(\mathsf{z},\ \ \lambda X_1 : \mathsf{exp}.\, X_1,\ \ \mathsf{nat},\ \ \mathsf{nat},\\ \qquad\qquad \mathsf{hasType\_z},\\ \qquad\qquad \lambda X_2 : \mathsf{exp}.\, \lambda D : \mathsf{deriv}.\, \mathsf{hasType\_s}\,(X_2,\ D))\end{array}$$

The last rule applied is **hasType_let**, as witnessed by the symbol hasType_let. The first four arguments (first line) correspond to the instantiation of its schematic variables (recall that the fact that x is bound in the body of "let" is rendered as a λ-abstraction in higher-order abstract syntax). The second line encodes a simple derivation of the left premise of that rule ($\cdot \vdash \mathsf{z} : \mathrm{nat}$). The last argument (third line) shows the representation of the parametric and hypothetical derivation $x : \mathrm{nat} \vdash \mathsf{s}\,(x) : \mathrm{nat}$: given an expression X_2 (for x) and a derivation D (showing that x has type nat), it constructs a derivation of the fact that $\mathsf{s}\,(X_2)$ has type nat. The type of the constant hasType_let is: $\mathsf{exp} \times (\mathsf{exp} \to \mathsf{exp}) \times \mathsf{tp} \times \mathsf{tp} \times \mathsf{deriv} \times (\mathsf{exp} \times \mathsf{deriv} \to \mathsf{deriv}) \to \mathsf{deriv}$.

8 Dependent Types

Hand-encoding derivations as part of the predicates that represent judgments had an arbitrary flavor already when we first used this technique in Section 4. With parametric and hypothetical judgments, it acquires a degree of complexity that makes the appearance of an instrumented clause rather cryptic and the likelihood of making mistakes quite high. Again, this goes against immediacy.

The structure of our encoding of derivations is however simple: we have one constant per clause, and its type is isomorphic to the structure of this formula. Compare for example the skeleton of the above clause for rule **hasType_let** (left) and the type of the associated constant hasType_let (right):

$$\begin{array}{l|l}\forall\_{:}\mathsf{exp}.\,\forall\_{:}\mathsf{exp}\to\mathsf{exp}.\,\forall\_{:}\mathsf{tp}.\,\forall\_{:}\mathsf{tp}. & \mathsf{exp}\times(\mathsf{exp}\to\mathsf{exp})\times\mathsf{tp}\times\mathsf{tp}\times\\ \quad \mathsf{hasType}\,\_ \wedge & \quad \mathsf{deriv}\times\\ \quad (\forall\_{:}\mathsf{exp}.\,\mathsf{hasType}\,\_ \Rightarrow \mathsf{hasType}\,\_) & \quad (\mathsf{exp}\times\mathsf{deriv}\to\mathsf{deriv})\\ \quad \Rightarrow \mathsf{hasType}\,\_ & \quad \to \mathsf{deriv}\end{array}$$

This observation suggests that the type of a derivation constant such as hasType_let can be computed from the associated clause. Going one step further, a clause ascertaining the derivability of a judgment can be augmented automatically to check derivations. Although an important realization, this is still very ad-hoc. We will instead go the other way around and enrich our language of types so that the declaration of a derivation constant will subsume the corresponding clause. This idea (as well as the above observation) rests on a fascinating correspondence between logic and type theory known as the *Curry-Howard isomorphism* [deGroote 1995]. We will now sketch how this can be achieved.

In the absence of errors, positively verifying that a given term D has type deriv ensures that D is the representation of a valid derivation. We may ask: "a derivation of what judgment?". We can easily refine our encoding to discriminate generically between typing and evaluation derivations, for example by replacing deriv with sorts hasType_deriv and eval_deriv. However, being more specific is not achievable in any simple way.

Upgrading our language of declarations to a *dependent type theory* enables us to verify whether a term D is a valid derivation of a judgment $\Gamma \vdash E : T$ for specific E and T, for example: indeed, it allows us to check whether the type of D is $\mathsf{hasType\_deriv}(E^*, T^*)$, where E^* and T^* are the encodings of E and T, respectively. Observe that hasType_deriv has now two arguments. The type of the constant hasType_let now assumes the following scary form:

$$\begin{array}{l}\Pi E_1{:}\mathsf{exp}.\,\Pi E_2{:}\mathsf{exp}\to\mathsf{exp}.\,\Pi T, T'{:}\mathsf{tp}.\\ \quad \mathsf{hasType\_deriv}\,(E_1,\ T')\\ \quad \times\,(\Pi X{:}\mathsf{exp}.\ \mathsf{hasType\_deriv}\,(X,\ T')\to \mathsf{hasType\_deriv}\,(E_2\ X,\ T))\\ \quad \to \mathsf{hasType\_deriv}\,(\mathsf{let}\,(E_1,\ E_2),\ T).\end{array}$$

where the *dependent type constructor* Π declares the type and the scope of variables. Informally, assigning this type to hasType_let specifies that, given encodings D_1 and D_2 for derivations of $\Gamma \vdash E_1 : T'$ and $\Gamma, x : T' \vdash E_2 : T$ respectively, the term $\mathsf{hasType\_let}\,(E_1, E_2, T, T', D_1, D_2)$ represents a derivation of $\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : T$.

Adopting a dependent type theory not only enables a finer classification of types, but it also allows us to dispense with the clausal representation of inference rules altogether. Observe indeed that the above type is isomorphic to the clause for **hasType_let**: simply replace hasType_deriv with hasType, Π with $\forall$, $\times$ with $\wedge$, and $\rightarrow$ with $\Rightarrow$. This fact results from the Curry-Howard isomorphism.

The type theory we briefly described is closely related to the language of the logical framework *LF* [Harper 1993]. It admits decidable type-checking and an interpretation as a higher-order logic programming language. This popular logical framework has been implemented as the *Twelf* environment [Twelf].

9 Further Issues

The logical framework we just finished sketching permits adequately and elegantly representing numerous formalisms. However, there are entire classes of deductive systems that cannot be given an immediate formalization in that language. Certain issues have been successfully addressed by enacting the refinement methodology presented in this paper. For example, formalisms that embed a notion of mutable state (e.g. procedural programming languages such as *C* and *Java*, databases, or process calculi) can be given adequate representations within logical frameworks based on a *linear type theory* [Cervesato 1996].

Other issues are the object of intensive research. Among these we should recall the representation of properties that apply to all individual elements of a given class (*extensional universal quantification*) and the related notion of non-derivability (*negation*). Another example is the concept of *ordered assumptions*.

References

[Cervesato 1996] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Proc. LICS'96*, pages 264–275, 1996.

[Constable 1986] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, 1986.

[deBrujn 1980] N. G. de Bruijn. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[deGroote 1995] Philippe de Groote, editor. *The Curry-Howard Isomorphism.* Academia, 1995.

[Dowek 1993] Gilles Dowek et al. The *Coq* proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993.

[Harper 1993] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journ. ACM*, 40(1):143–184, 1993.

[Martin-Löf 1985] Per Martin-Löf. Truth of a proposition, evidence of a judgment, validity of a proof. Notes to a talk given at the workshop *Theory of Meaning*, Florence, Italy, 1985.

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

[Miller] Dale Miller. Lambda Prolog: An introduction to the language and its logic. Forthcoming; current draft available from `http://cse.psu.edu/~dale/lProlog`.

[Miller 1991] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Paulson 1993] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, 1993.

[Pfenning 1991] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Sterling 1994] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1994. Second edition.

[Twelf] Frank Pfenning and Carsten Schürmann. The *Twelf* project. `http://www.cs.cmu.edu/~twelf/`.