December 1983

Report No. STAN-E-83-992

The Language of an Interactive Proof Checker

by

Jussi Kctonen and Joseph S. Weening

Department of Computer Science

Stanford University
Stanford, CA 94305

The Language of an Interactive Proof Checker

Jussi Ketonen
and
Joseph S. Weening

Stanford University

1. Abstract

We describe the underlying language for EKL, an interactive theorem-proving system currently under development at the Stanford Artificial Intelligence Laboratory. Some of the reasons for its development as well as its mathematical properties are discussed.

Research supported by NSF grant MCS-82-06565 and ARPA contract N00039-82-C-0250.

2. Introduction

This paper addresses the problem of providing **a** mechanizable language for mathematical reasoning-a domain characterized by highly abbreviated symbolic manipulations whose logical complexity tends to be rather low.

A word about our goals. Our primary intention is not to present difficult results in an established part of mathematical logic. Instead, we will exhibit a formal framework which we have found to be particularly useful in **mechanising** mathematical reasoning. Hopefully, our approach will highlight some of the problems (and possible solutions) encountered in this endeavor. The study of logical properties of our formalism is only incidental to demonstrating it as a sound and expressive vehicle for mathematical reasoning.

The reader may be surprised that we consider the problem of formally representing mathematical knowledege to be open; after all, first-order logic certainly provides ***an adequate*** framework for mathematics. However, much of the ***structure*** of mathematical knowledge is lost in the process of translation into first-order logic. This problem becomes particularly acute when one tries to represent schematic knowledge in a mechanizable way. We could beg the question and declare all knowledge not treatable in a first-order framework ***to*** be ***meta-theoretic*** **in nature;** what is meta-theoretic depends, after all, on one's point of view. We do not find this particularly satisfactory; as a disguised form of programming, meta-theory may not present any apparent increase in either correctness or clarity.

While first order logic does not offer **a** smooth mechanization of mathematical reasoning, it can serve as an adequate basis to start from. We want to preserve its basic proof-theoretic properties. However, our primary criterion is that of ***expressibility*** and the ability to talk about the intrinsic properties of the concepts in question. Such accidents of formalization as decidability (or the complexity of a decision procedure) are of only secondary importance; we recognize the fact that as of now we know of very few decidable theories that correspond to natural fragments of mathematics. Since even the simplest known decision procedure for a non-trivial part of logic has an exponential worst-case performance, it is more useful for us to tailor our algorithms to

naturally occurring inputs rather than use arbitrary syntactic constraints. Indeed, we expect such trivial syntactic criteria as the number of quantifiers in a formula to appeal only to the most technically oriented logician.

Most of our discussion takes place in the context of EKL, an interactive theorem-proving system under development at Stanford. EKL can verify the correctness of proofs in various branches of mathematics. To date, EKL has been most heavily used in the areas of computer program correctness, logic, and the foundations of mathematics. From these experiments, we have discovered a formal language which is highly flexible and able to express statements in a natural way in all of these areas. At the same time it is a relatively minor extension of standard logical concepts-EKL can be viewed as a typed high-order logic.

We will describe this language and its semantics. The (preferred) semantic interpretation of the language is of primary importance; hence we base the development of the language in this paper on this interpretation, rather than give a set of axioms that the theory must satisfy. We have adopted a strongly *extensional* point of view-our language is *not* intended as a model for computation.

The two main concepts introduced are *terms* and *types.* Each is presented as a class of objects in a set-theoretic universe. Terms represent the objects of the mathematical proof; they are the base elements, functions, relations, etc. as used by the mathematician. Types give us a way of restricting the class of acceptable formulas in the language, to prevent logical contradictions from occurring. Formulas and terms are treated in a completely uniform manner; formulas are simply terms of type **truthval**. Several operations on types are presented, and the algebraic structure of the class of types is investigated. Of particular interest is the notion of *list types,* which allows us to talk in a natural way about parameterized formulas and functions taking an arbitrary number of variables.

Using these ideas, we show how logical sentences and proofs are represented in EKL, and demonstrate the soundness of this representation. Finally, the meta-theoretic extensions to EKL are presented. They allow us to talk about *names* of objects and axiomatize the formal semantics of EKL. Our reason for the introduction of meta theory to EKL is to tie the formalism to its

intended (preferred) interpretation and *to parametrize* the logic by allowing arbitrary sets and functions as objects of discourse. Instances of deduction can then be replaced by computation in the "real world".

This paper is partly intended as an abstract implementation guide for a language for theorem-proving.

We would like to thank John McCarthy, Carolyn Talcott and Richard Weyhrauch for many stimulating discussions.

3. Tupling

We imagine working in a reasonable facsimile of the set-theoretic universe. For our purposes it is sufficient to assume that *we* have a *concrete Cartesian closed category* (c.c.c) C with a few additional properties. A detailed definition of Cartesian closed categories is given in [Scott 1980].

We assume that C is full subcategory of the category of all sets. Thus, for any two objects a, b of C, the set Hom(a, b) is the set of all functions from a to b. By the c.c.c. assumption, C is closed under Cartesian products. We will require more; we ask that C be closed under countable unions. C is also assumed to contain the distinguished elements **true, false, Boole = {true, false}** and **()**, the empty sequence.

Since C is Cartesian closed, we have the notion of *evaluation*, the map $f, x \mapsto f(x)$; and λ-*abstraction*, the adjunction Hom($a \times b$, c) $\simeq$ Hom(a, Hom(b, c)).

In short, we can view C as a rather standard model of typed X-calculus. However, in order to deal with functions admitting an arbitrary number of variables, we will modify C further. Consider the smallest equivalence relation $\approx$ closed under

$$(_{\times}) \approx x,$$

$$(x_1, x_2, \ldots, x_n, (y_1, y_2, \ldots, y_m)) \approx (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m),$$

$$(x_1, x_2, \ldots, x_n, ()) \approx (x_1, x_2, \ldots, x_n),$$

$$x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n \supset (x_1, x_2, \ldots, x_n) \approx (y_1, y_2, \ldots, y_n).$$

Let $\mathcal{S}$ be the subcategory of C consisting of the objects of C together with all morphisms f of C which are well-behaved under $\approx$:

$$x \approx y \supset f(x) \approx f(y).$$

Let V be the category whose objects are the closures of the objects of $\mathcal{S}$ under $\approx$ and whose morphisms are those induced by the morphisms of $\mathcal{S}$. It can again be realized as a full subcategory of the category of all sets. The quotient map induces a natural transformation $q : \mathcal{S} \to \mathcal{V}$. However, the natural notion of "product,, inherited from C via the quotient map q coincides no longer with the categorical product. For simple structures, q induces an isomorphism. For example, if neither a nor b contain sequences of the form $(x_1, x_2, \ldots, x_n)$, where either $n = 1$ or one of the x_i is a sequence or $x_n = ()$, then $\mathbf{Hom}_C(a, b) \simeq \mathbf{Hom}_{\mathcal{S}}(q(a), q(b))$. For the sake of simplicity, we will go ahead and identify elements with their equivalence classes, and use standard notation for the quotient of the product operation, even though its functorial properties are different.

It should be noted that the fullness of V is not really a necessary formal criterion for our theory to go through. What is actually needed is that the type algebra **TA** defined in section 7 contain infinitely many indecomposable elements in the sense to be explained later.

The category $\mathcal{S}$ is still closed under X-abstraction. For any a, b, c the adjoint functor described above induces injections

$$\mathrm{Hom}(a \times b, c) \to \mathbf{Hom}(a, \mathrm{Hom}(b, c)),$$
$$\mathbf{Hom}(a \times b, c) \to \mathrm{Hom}(b, \mathbf{Hom}(a, c)).$$

FACT 3.1.: For all elements of V:

$$(x) = x,$$
$$(x_1, x_2, \ldots, x_n, (y_1, y_2, \ldots, y_m)) = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m),$$
$$(x_1, x_2, \ldots, x_n, ()) = (x_1, x_2, \ldots, x_n),$$
$$(x, u) = (y, v), u \neq (), v \neq () \supset x = y \wedge u = v.$$

The tupling operation can be viewed as a way of encoding finite sequences that is right-associative and has an "empty,, element () that acts as a right unit.

DEFINITION 3.1.: An element x is *tupled* if x = (y, z) for some y and some $z \neq ()$.

DEFINITION 3.2.: A sequence $x_1, x_2, \ldots, x_n$ is *in tuple normal form if* $x_n \neq ()$ and x_n is not tupled.

Using induction on tuples, we can easily prove the following.

FACT 3.2.: If x is tupled, then there is a unique sequence $x_1, x_2, \ldots, x_n$ in tuple normal form such that $x = (x_1, x_2, \ldots, x_n)$.

The uniqueness of tuple normal form implies the existence of operations that are inverse to tupling, namely *projections*. We define two kinds of projections for each $i \geq 1$: π_i, the projection onto the ith coordinate of a tuple, and πl_i, the projection onto the ith "tail.,,

DEFINITION 3.3.: If x is not tupled, then for all $i \geq 1$,

$$\pi_i(x) = x, \quad \text{and} \quad \pi l_i(x) = x.$$

If $x_1, x_2, \ldots, x_n$ is a sequence in tuple normal form, then for all $i \geq 1$,

$$\pi_i((x_1, x_2, \ldots, x_n)) = \begin{cases} x_i, & \text{if } 1 \leq i \leq n; \\ x_n, & \text{otherwise,} \end{cases}$$

and

$$\pi l_i((x_1, x_2, \ldots, x_n)) = \begin{cases} (x_{i+1}, \ldots, x_n), & \text{if } 1 \leq i < n; \\ x_n, & \text{otherwise.} \end{cases}$$

These definitions easily imply the following facts.

FACT 3.2.: For any sequence $x_1, x_2, \ldots, x_n$ with $x_n \neq ()$, and any $i \geq 1$,

$$\pi_i((x_1, x_2, \ldots, x_n)) = \begin{cases} x_i, & \text{if } 1 \leq i < n; \\ \pi_{i-n+1}(x_n), & \text{otherwise.} \end{cases}$$

$$\pi l_i((x_1, x_2, \ldots, x_n)) = \begin{cases} (x_{i+1}, \ldots, x_n), & \text{if } 1 \leq i < n; \\ \pi l_{i-n+1}(x_n), & \text{otherwise.} \end{cases}$$

FACT 3.3.: For all i, j,

$$\pi_i \circ \pi l_j = \pi_{i+j},$$

$$\pi l_i \circ \pi l_j = \pi l_{i+j}.$$

DEFINITION 3.4.: For any class A, A^* denotes the set of sequences of elements of A, namely,

$$A^* = \{()\} \cup \bigcup_{n \geq 1} \{(a_1, \ldots, a_n) : a_i \in A \text{ for } 1 \leq i \leq n\}.$$

4. The Language of Pure EKL

As a part of the universe $\mathcal{V}$ we postulate the language $\mathcal{L}$ of EKL. It need not be a recursive set-or a set at all. $\mathcal{L}$ consists of three classes: the class of all terms, $\mathcal{T}$, the class of ordinary atoms, A, and the class of distinguished atoms, D. The class D consists of names for

$$\supset, \equiv, \vee, A, \neg, cond, tuple, \pi_1, \pi_2, \ldots, \pi l_1, \pi l_2, \ldots, =, \neq, universal, \forall, 3, \lambda, empty, true, false.$$

We use A^+ to denote the class of all atoms: $A^+ = \mathcal{A} \cup D$. The class $\mathcal{T}$ contains A+. To construct terms, we have operations

$$Make\text{-}application : \mathcal{T} \times \mathcal{T}^* \to \mathcal{T} - \mathcal{A}^+,$$

$$Make\text{-}exist, Make\text{-}universal, Make\text{-}lambda : A^* \times \mathcal{T} \to \mathcal{T} - \mathcal{A}^+.$$

Note that atoms are not in the ranges (which are assumed to be disjoint) of *Make-application, Make-exist, Make-universal* and *Make-lambda.* $\mathcal{T}$ is the closure of $\mathcal{A}^+$ under these operations. Thus we can prove facts about $\mathcal{T}$ using standard inductive arguments. In sections 5 to 8 we will consider only the class of all quantifier-free terms, $\mathcal{T}_0$; the closure of $\mathcal{A}^+$ under *Make-avvlication.* For the sake of simplicity, we use the notations

$f(x_1, x_2, \ldots, x_n)$	for $Make\text{-}application(j, (x_1, x_2, \ldots, x_n))$,
$3x_1 \ldots x_n.t$	for $Make\text{-}exist((x_1, x_2, \ldots, x_n), t)$,
$\forall x_1 \ldots x_n.t$	for $Make\text{-}universal((x_1, x_2, \ldots, x_n), t)$,
$xx_1 \ldots x_n.t$	for $Make\text{-}lambda((x_1, x_2, \ldots, x_n), t)$,
if x then y else z	for $cond(x, y, x)$,
0	for $empty$,
$(x_1, x_2, \ldots, x_n)$	for $tuple(x_1, x_2, \ldots, x_n)$,

when the situation affords an unambiguous interpretation. In addition, we regard atoms like $\equiv$, $=$, $\neq$ as infix operators and atoms like $\neg$ as unary operators; i.e., we write $\neg x$ for $Make\text{-}application(\neg, (x))$.

Make-application, Make-univetaal, Make-exist and ***Make-lambda*** are assumed to be **injective** functions. In fact, there are partial operators ***operator-of, operand.+of, bindingtuple-of*** and ***matrix-of,*** with obvious commutativity rules:

$$operator\text{-}of(Make\text{-}application(x, y)) = x,$$
$$operands\text{-}of(Make\text{-}application(x, y)) = y,$$
$$operator\text{-}of(Make\text{-}exist(y, z)) = 3,$$
$$operator\text{-}of(Make\text{-}universal(y, z)) = \forall,$$
$$operator\text{-}of(Make\text{-}lambda(y, z)) = \lambda,$$
$$bindingtuple\text{-}of(Make\text{-}universal(y, z)) = y,$$
$$matrix\text{-}of(Make\text{-}universal(y, z)) = z,$$
$$bindingtuple\text{-}of(Make\text{-}exist(y, z)) = y,$$
$$matrix\text{-}of(Make\text{-}exist(y, z)) = z,$$
$$bindingtuple\text{-}of(Make\text{-}lambda(y, z)) = y,$$
$$matrix\text{-}of(Make\text{-}lambda(y, z)) = z.$$

5. Semantics: Interpetation of Quantifier-free EKL Terms

The evaluation of EKL terms takes place in the category $\mathcal{V}$. The behavior of the "EKL interpreter" on the class of quantifier-free terms $\mathcal{T}_0$ can then be described using the following rules.

DEFINITION 5.1.: An ***environment*** is a partial map j : A $\rightarrow \mathcal{V}$. Let Env denote the class of all environments. For two environments j, g define their concatenation j#g as follows.

$$\mathrm{Dom}\, f \# g = \mathrm{Dom}\, f \cup \mathrm{Dom}\, g,$$

(j#g)(x) = if x $\in$ Dom j then j(x) else g(x).

DEFINITION 5.2.: Each j $\in$ Env induces a ***partial valuation map*** Eval(j) : $\mathcal{T}_0 \rightarrow \mathcal{V}$ as follows.

$\mathbf{Eval}(f)(\text{true}) = \mathbf{true},$

$\mathbf{Eval}(f)(\mathit{false}) = \mathbf{false},$

$\mathbf{Eval}(f)(\mathit{empty}) = (),$

$\mathbf{Eval}(f)(t) = f(t), \quad \text{if } t \in \mathcal{A},$

$\mathbf{Eval}(f)(\mathit{tuple}(x_1, x_2, \ldots, x_n)) = (\mathbf{Eval}(f)(x_1), \ldots, \mathbf{Eval}(f)(x_n)),$

$\mathbf{Eval}(f)(\pi_i(x_1, x_2, \ldots, x_n)) = \pi_i(\mathbf{Eval}(f)(x_1), \ldots, \mathbf{Eval}(f)(x_n)), \quad i = 1, 2, \ldots,$

$\mathbf{Eval}(f)(\pi l_i(x_1, x_2, \ldots, x_n)) = \pi l_i(\mathbf{Eval}(f)(x_1), \ldots, \mathbf{Eval}(f)(x_n)), \quad i = 1, 2, \ldots.$

Note that π_i and πl_i have different meanings on the left and right sides of the last two equations above. On the left side they are members of $\mathcal{L}$; on the right, they are the operations discussed in section 3.

The interpretations of the operations $\wedge$, $\vee$, $\neg$, $\supset$ and $\equiv$ follow standard rules. For example, if $\mathbf{Eval}(f)(x), \mathbf{Eval}(f)(y) \in \mathbf{Boole}$, then

$$\mathbf{Eval}(f)(x \wedge y) = \begin{cases} \mathbf{true}, & \text{if } \mathbf{Eval}(f)(x) = \mathbf{Eval}(f)(y) = \mathbf{true}; \\ \mathbf{false}, & \text{otherwise.} \end{cases}$$

For $x, y, z \in \mathcal{T}$,

$$\mathbf{Eval}(f)(x = y) = \begin{cases} \mathbf{true}, & \text{if } \mathbf{Eval}(f)(x) = \mathbf{Eval}(f)(y); \\ \mathbf{false}, & \text{otherwise,} \end{cases}$$

$\mathbf{Eval}(f)(\mathit{universal(z)}) = \mathbf{true},$

$$\mathbf{Eval}(f)(\mathbf{if}\ x\ \mathbf{then}\ y\ \mathbf{else}\ z) = \begin{cases} \mathbf{Eval}(f)(y), & \text{if } \mathbf{Eval}(f)(x) = \mathbf{true}; \\ \mathbf{Eval}(f)(z), & \text{if } \mathbf{Eval}(f)(x) = \mathbf{false}. \end{cases}$$

If h is not a distinguished symbol,

$$\mathbf{Eval}(f)((x_1, x_2, \ldots, x_n)) = (v_1, v_2, \ldots, v_n),$$

and $(v_1, v_2, \ldots, v_n)$ is an element of the source of the function $\mathbf{Eval}(f)(h)$, then

$$\mathbf{Eval}(f)(h(x_1, x_2, \ldots, x_n)) = [\mathbf{Eval}(f)(h)](v_1, v_2, \ldots, v_n).$$

DEFINITION 5.3.: Given a term $t \in \mathcal{T}_0$, its ***content C(t) is*** the class

$$\{(f, \mathbf{Eval}(f)(t)) : t \in \mathrm{Dom}\ \mathbf{Eval}(f), f \in \mathbf{Env}\}.$$

DEFINITION 5.4.: A term $t \in T_0$ is ***meaningful*** if ***C(t)*** is non-empty.

6. Stability: **Problems with the Notion of Interpretation**

Consider the following terms:

$$f(f)$$

$$f \textit{ (if true then true else } f)$$

$$h(f(0)).$$

The first clearly has no meaning. The second case is more interesting; it has meaning, namely it evaluates *to* the value of f *(true)* if true belongs to the domain of the value of f. However, a small perturbation in the syntax of the formula (namely, replacing *true* with *false*) causes it to have no meaning at all. We are interested in the third term under the following interpretation: h is the function on natural numbers such that $h(n) = n + 1$ and f is the function

$$f(x) = \begin{cases} 0, & \text{if } x = 0; \\ \text{apples}, & \text{if } x = 1; \\ \text{oranges}, & \text{otherwise.} \end{cases}$$

Clearly, $h(f(0))$ is meaningful under this interpretation. However, if we were to replace f with a different function with the same source and target we could easily end up with a no interpretation at all.

Finally, the interpretation of of a term of the form $f(x)$ is sensitive to inclusion relations: If f is interpreted as a function with source A, and x is regarded as an element of a set $B \subseteq A$, then we would like to interpret x via the *inclusion map* $i_{BA} : B \to A$ so that $f(x)$ makes sense. The current definition of semantics does not allow this.

These examples exhibit ***instabilities*** of the definition of interpretation of a term; instabilities in the sense that small perturbations in the interpretation of a term or its parts may cause lack of meaning.

To make the notion of stability more precise, let $\simeq$ be the smallest equivalence relation closed under the following rules:

$$\mathbf{true} \simeq \mathbf{falae};$$
$$x_1 \simeq y_1 \wedge \cdots \wedge x_n \simeq y_n \supset (x_1, x_2, \ldots, x_n) \simeq (y_1, y_2, \ldots, y_n);$$
$$f, g \in \mathbf{Hom}(x, y) \supset f \simeq g.$$

DEFINITION *6.1.:* **A** term $t \in \mathcal{T}_0$ ***is stable*** under an interpretation $f : A \rightarrow \mathcal{V}$ if and only if all the immediate subterms, say $t_1, t_2, \ldots, t_n$, of t are stable under f and there is a sequence of inclusion maps $(i_1, i_2, \ldots, i_n)$ such that for for any set $x_1, x_2, \ldots, x_n$ with $x_k \simeq \mathbf{Eval}(f)(t_k)$, t has a value if we use $i_k(x_k)$ instead $\mathbf{Eval}(f)(t_k)$ in the value computation. We also say that f *is* a ***stable interpretation*** of t.

DEFINITION 6.2.: $|X|$ denotes {y : y $\simeq$ x for some x $\in$ X}. $E(f)$ denotes the function $\lambda x.\, |\{f(x)\}|$.

DEFINITION 6.3.: For any stable interpretation f of a term $t \in \mathcal{T}_0$, the ***type*** of t under f is the equivalence class of all values of $\mathbf{Eval}(f)(t)$ under the perturbations described above.

LEMMA 6.1.: For any stable interpretation f of a term $t \in \mathcal{T}_0$, the type of t under f depends only on $E(f)$.

7. **The Algebra** of Types

To determine whether a term t is meaningful by computing all its potential interpretations is hard. However, it is easy to compute the equivalence classes of contents of terms under stable interpretations by using algebraic properties of types. In fact we can provide a simple way of deciding whether a term has a stable interpretation.

DEFINITION 7.1.: **A** ***type*** is a non-empty set that is a =-equivalence class.

Define operators $\otimes$, $\rightarrow$, $\vee$, $*$, π_i and πl_i on types as follows.

$$A \otimes B = |A \times B|,$$
$$A \rightarrow B = |\mathrm{Hom}(A, B)|,$$
$$A \vee B = |A \cup B|,$$
$$A^* = |A^*|,$$
$$\pi_i(A) = |\{\pi_i(x) : x \in A\}|,$$
$$\pi l_i(A) = |\{\pi l_i(x) : x \in A\}|.$$

Let **empty** $= |\{()\}|$ and **truthval** $= |\{\mathbf{true}, \mathbf{false}\}|$. Define a partial order $\leq$ on the class of all types by setting $A \leq B$ if and only if $A \subseteq B$. We adjoin an ***error term*** $\perp$ with the following proper ties:

$$A \otimes \perp = \perp, \quad \perp \otimes A = \perp, \quad (A \rightarrow \perp) = \perp, \quad (\perp \rightarrow A) = \perp,$$
$$A \vee \perp = \perp, \quad \perp \vee A = \perp, \quad \perp^* = \perp.$$

In particular, for any type A, $A \leq \perp$.

Note that the operation $\vee$ intuitively represents the ***union*** operation, not disjoint union usually present in standard typing systems. This allows us far more flexibility for typing: For example, conditional statements have a much wider range of applicability

DEFINITION 7.2.: The ***type algebra*** TA is the class of all types augmented by the structure given above and special elements **empty, truthval** and $\perp$.

It can be shown that TA forms an ***error*** algebra in the sense of [Goguen **1978**].

THEOREM 7.1.: TA satisfies the following properties:

(1) $\otimes$ is a right associative operator with the right unit ***empty.***

(2) $\leq$ is a partial order derived from the associative, commutative and idempotent operator $\vee$: $x \leq y$ **if and only if** $x \vee y = y$.

(3) $\vee$ is distributive with respect to $\otimes$.

(4) For any a: ***empty*** $\leq a^*$ and $a \otimes a^* \leq a^*$.

(5) ***If*** $a = b \rightarrow c$, then $\pi_i(a) = a$ and $\pi l_i(a) = a$.

(6) If $a = a_1 \vee \cdots \vee a_n$, **then** $\pi_i(a) = \pi_i(a_1) \vee \cdots \vee \pi_i(a_n)$ **and** $\pi l_i(a) = \pi l_i(a_1) \vee \ldots \vee \pi l_i(a_n)$. If $a \leq b$, then $\pi_i(a) \leq \pi_i(b)$ and $\pi l_i(a) \leq \pi l_i(b)$.

(7) If $t = t_1 \otimes \cdots \otimes t_n$ and $u = t_1 \otimes \cdots \otimes t_{n-1}$, **then**

$$\pi_i(t) = \begin{cases} t_i, & \text{if } i < n,\ t_n \not\geq \mathit{empty}; \\ t_i \vee \pi_i(u), & \text{if } i < n,\ t_n > \mathit{empty}; \\ \pi_{i-n+1}(t_n), & \text{if } i \geq n,\ t_n \not\geq \mathit{empty}; \\ \pi_{i-n+1}(t_n) \vee \pi_i(u), & \text{if } i \geq n,\ t_n > \mathit{empty}, \end{cases}$$

and

$$\pi l_i(t) = \begin{cases} t_{i+1} \otimes \cdots \otimes t_n, & \text{if } i < n,\ t_n \not\geq \mathit{empty}; \\ t_{i+1} \otimes \cdots \otimes t_n \vee \pi l_i(u), & \text{if } i < n,\ t_n > \mathit{empty}; \\ \pi l_{i-n+1}(t_n), & \text{if } i \geq n,\ t_n \not\geq \mathit{empty}; \\ \pi l_{i-n+1}(t_n) \vee \pi l_i(u), & \text{if } i \geq n,\ t_n > \mathit{empty}. \end{cases}$$

(8) If $t = a^*$, **then**

$$\pi_i(t) = \begin{cases} \mathit{empty} \vee \pi_1(a) \vee a, & \text{if } i = 1; \\ \mathit{empty} \vee \pi_i(a) \vee \pi_{i-1}(t), & \text{if } i > 1, \end{cases}$$

and

$$\pi l_i(t) = \begin{cases} \mathit{empty} \vee \pi l_1(a) \vee a^*, & \text{if } i = 1; \\ \mathit{empty} \vee \pi l_i(a) \vee \pi l_{i-1}(t), & \text{if } i > 1. \end{cases}$$

(9) For all i, j, $\pi_i \circ \pi l_j = \pi_{i+j}$ and $\pi l_i \circ \pi l_j = \pi l_{i+j}$.

(10) If a, b, c, $d \neq \bot$ then there is **no type** e such that $e \leq a \to b$ and $e \leq c \otimes d$.

(11) For any $a, b \neq \bot$, $a \to b \leq c \to d$ if and only if $a = b$ and $c = d$.

(12) Call a type a ***is minimal*** if for all $b \leq a$, $b = a$ and ***indecomposable*** if it is minimal and for no c, d, $(c \to d) \leq a$ or $(c \otimes d) \leq a$ with $d \neq \mathit{empty}$. **The** types **truthval** and **empty are** indecomposable. Furthermore, for all indecomposable a and $i \geq 1$, $\pi_i(a) = a$ and $\pi l_i(a) = a$. If a is indecomposable, then for no c, d, $(c \to d) \geq a$ or $(c \otimes d) \geq a$ with $d \not\geq \mathit{empty}$.

Let $\mathcal{3}$ be the formal (first-order) system given by **the** language of the type algebra, with the properties given above as its axioms.

Clearly, a type a is minimal if and only if a is of the form $|\{x\}|$ for some x in $\mathcal{V}$.

Define two more operations on the set **TA as** follows. If the type ***A*** contains only functions with the same source C, set

$$\mathbf{dom}\, A = |C|,$$

otherwise let $\mathbf{dom}\, A = \bot$. Similarly, if ***D*** is the target of all of the functions in ***A,*** set

$$\mathbf{range}\, A = |D|.$$

If $c = (a \to d)$ and $c \neq \bot$, then $\mathbf{dom}\, c = a$, and $\mathbf{range}\, c = d$.

Next we shall define through mutual recursion the notions of ***minimal*** and ***primitive*** terms t in 3, together with a list $\mathbf{Conds}(t)$ of formulae in 3 of the form $v \not\geq \mathit{empty}$, where v is a variable.

DEFINITION 7.3.:

(1) If t is an atomic term, then t is minimal and $\mathbf{Conds}(t) = \emptyset$.

(2) If $t_1, t_2, \ldots, t_n$ is a sequence of minimal terms such that t_n *is* not the term ***empty*** and not a product, then $t = t_1 \otimes \ldots \otimes t_n$ is a minimal term with

$$\mathbf{Conds}(t) = \begin{cases} \{t_n \not\geq \mathit{empty}\} \cup \bigcup\{\mathbf{Conds}(t_i) : 1 \leq i \leq n-1\}, & \text{if } t_n \text{ is a variable;} \\ \bigcup\{\mathbf{Conds}(t_i) : 1 \leq i \leq n\}, & \text{otherwise.} \end{cases}$$

(3) If $t_1, t_2, \ldots, t_n$ is a sequence of minimal terms then $t = t_1 \vee \ldots \vee t_n$ is a primitive term **with**

$$\mathbf{Conds}(t) = \bigcup\{\mathbf{Conds}(t_i) : 1 \leq i \leq n\}.$$

(4) If t_1, t_2 are primitive terms, then $t = t_1 \to t_2$ **is** a minimal term such that $\mathbf{Conds}(t) = \mathbf{Conds}(t_1) \cup \mathbf{Conds}(t_2)$.

When **the list** $\mathbf{Conds}(t)$ is non-empty, we shall often confuse it with the conjunction $\bigwedge \mathrm{Conds}(t)$.

DEFINITION 7.4.: For any term t, its list of ***restricted variables*** $\mathbf{Vars}(t)$ is computed as follows.

(1) If t is an atomic term, then $\mathbf{Vars}(t) = \{t\}$.

(2) If $t = t_1 \vee \ldots \vee t_n$ or $t = t_1 \otimes \ldots \otimes t_n$, then $\mathbf{Vars}(t) = \bigcup\{\mathbf{Vars}(t_i) : 1 \leq i \leq n\}$.

(3) If $t = u^*$, $t = \pi_i(u)$ or $t = \pi l_i(u)$, then $\mathbf{Vars}(t) = \mathbf{Vars}(u)$.

(4) If $t = t_1 \to t_2$ then $\mathbf{Vars}(t) = \emptyset$.

LEMMA 7.1.: If t is a term not involving $*, \pi_i, \pi l_i$, then there is a primitive term t' such that $\mathcal{F} \vdash t = t'$.

PROOF: By induction on the term t.

LEMMA 7.2.: If t is a primitive term in variables $x_1, x_2, \ldots, x_n$ and for some types al, $a_2, \ldots, a_n$ satisfying $\mathbf{Conds}(t)$

$$\text{TA} \models t[a_1, a_2, \ldots, a_n] = \mathit{empty},$$

then t is the term empty. In particular, $\mathcal{3}\ \mathit{l\text{-}t} = \mathit{empty}$.

PROOF: By induction on the term t.

The following result shows that any minimal type can be represented by minimal terms and indecomposable types.

LEMMA 7.3.: If t = $t(x_1, x_2, \ldots, x_n)$ is a minimal term and $a_1, a_2, \ldots, a_n$ a sequence of types such that a_i is minimal for any x_i in $\mathbf{Vars}(t)$, then $t[a_1, a_2, \ldots, a_n]$ is minimal. Conversely, if a is a minimal type in TA, then we can construct in a canonical way distinct types and a minimal term t = $t(x_1, x_2, \ldots, \text{x},)$ such that $a_1, a_2, \ldots, a_n$ satisfies $\mathbf{Conds}(t)$ and

$$\text{TA} \models a = t[a_1, a_2, \ldots, a_n],$$

where each type a_i is indecomposable for x_i in $\mathbf{Vars}(t)$.

We shall denote this term t by Term(u). Lemma 7.5. will show that it is uniquely determined by its properties.

LEMMA 7.4.: If t is a minimal term and $t_1, t_2, \ldots, t_n$ is an arbitrary list of terms, then the following formula is true in TA when the ranges of the free variables of t in $\mathbf{Vars}(t)$ are restricted to minimal types:

$$t \leq (t_1 \vee \cdots \vee t_n) \equiv t \leq t_1 \vee \cdots \vee t \leq t_n.$$

Call a set $\{a_1, a_2, \ldots, a_n\}$ of types ***independent*** if no a_i is $\leq$ or $\geq$ to a **type** formed from the other a_j $(j \neq i)$, using the operations $\otimes, \rightarrow$ or $\vee$. Such sets are easy to construct; one could, for example, choose a_i to be distinct indecomposable types.

LEMMA 7.5.: Assume that t, u are minimal terms in variables $x_1, x_2, \ldots, x_n$ and $\{a_1, a_2, \ldots, a_n\}$ is an independent set of types satisfying $\mathbf{Conds}(t), \mathbf{Conds}(u)$ such that

$$\mathrm{TA} \models t[a_1, a_2, \ldots, a_n] \leq u[a_1, a_2, \ldots, a_n].$$

Then t = u as terms. In particular, $\mathcal{I} \vdash t = u$ if and only if $t = u$.

PROOF: By induction on (t, u).

TIIEOREM 7.2.: If t, u are any terms in free variables $x_1, x_2, \ldots, x_n$ not involving $^*, \pi_i, \pi l_i$, then

$$\mathbf{TA} \models \forall x_1 x_2 \ldots x_n . t = u \text{ if andonlyif } \mathcal{I} \vdash t = u.$$

In fact, 3 I- t = u if and only if $\mathrm{TA} \models t[a_1, a_2, \ldots, a_n] = u[a_1, a_2, \ldots, a_n]$ for some independent set $\{a_1, a_2, \ldots, a_n\}$ of minimal types satisfying $\mathbf{Conds}(t) \cup \mathbf{Conds}(u)$. Similarly for inequalities of the form $t \leq$ u.

PROOF: It suffices to prove by induction the above fact for $t \leq$ u, where t, u are primitive types. The inductive stage follows from Theorem 7.1. and Lemmas **7.1.–7.5.**

If t and u are primitive we can say more; t and u have to be of the form $t_1 \vee \cdots \vee t_n$, $u_1 \vee \ldots \vee u_m$ respectively, where t_i, u_j are minimal. It follows that

$$\mathcal{I} \vdash t = u$$

if and only if for all i there is a j such that $\mathcal{I} \vdash t_i = u_j$ and vice versa. Combining this with Lemma **7.5.**, we get a simple decision procedure for deciding whether the formula t = u is identically true in TA for any terms t, u not involving $^*, \pi_i$ or πl_i.

These methods can be extended to terms using $\pi_i, \pi l_i$ and *. To do this, we need to modify Definition 7.3.:

DEFINITION 7.5.: The notion of *-minimality and *-primitivity is defined as follows.

(1) If t is an atomic term, then t is *-minimal.

(2) If t is of the form $f_1(f_2(\cdots f_k(v)\cdots))$ where every f_j is of the form π_i or πl_i for some i, v a variable and only f_1 can be of the form πl_i, then t is *-minimal and $\mathbf{Conds}(t) = \emptyset$.

(3) If $t_1, t_2, \ldots, t_n$ is a sequence of *-minimal terms such that t_n is not the term *empty* and not a product, then $t = t_1 \otimes \ldots \otimes t_n$ is *-minimal. If t is a variable v or of the form $f_1(f_2(\cdots f_k(v)\cdots))$, then

$$\mathbf{Conds}(t) = \{v \not\geq \mathit{empty}\} \cup \bigcup\{\mathbf{Conds}(t_i) : 1 \leq i \leq n-1\},$$

otherwise

$$\mathbf{Conds}(t) = \bigcup\{\mathbf{Conds}(t_i) : 1 \leq i \leq n\}.$$

(4) If $t_1, t_2, \ldots, t_n$ is a sequence of *-minimal terms, then $t = t_1 \vee \ldots \vee t_n$ is a *-primitive term.

(5) If t is *-primitive and $t \neq \mathit{empty}$, then t^* is *-minimal and $\mathbf{Conds}(t^*) = \mathbf{Conds}(t)$.

(6) If t_1, t_2 are *-primitive terms, then $t = t_1 \to t_2$ is a *-primitive term.

LEMMA 7.6.: For any term t, there is a sequence of *-primitive terms $(t_1, t_2, \ldots, t_n)$ and formulas $(\phi_1, \phi_2, \ldots, \phi_n)$ such that each ϕ_i is a conjunction of formulas of the form $v \not\geq \mathit{empty}$, $v > \mathit{empty}$ and $v = \mathit{empty}$ (v a variable), with

$$\mathcal{F} \vdash \phi_1 \vee \cdots \vee \phi_n,$$

and for all i,

$$\mathcal{F} \vdash \phi_i \supset t = t_i.$$

PROOF: By induction on t. We can "push,, the projections in by using Theorem 7.1. This **generates** conditions of the form $t > \mathit{empty}$, $t \not\geq \mathit{empty}$ and $t = \mathit{empty}$, where t is a term. These conditions can be further reduced by applying the indecomposability of *empty*. The final requirements on the order of applications of the projections are satisfied by part (10) of Theorem 7.1.

THEOREM 7.3.: If $t = t(x_1, x_2, \ldots, x_n)$ is any term not involving *, then one can in a uniform way construct a primitive term

$$t_\sigma = t_\sigma(\bar{y}^1, \ldots, \bar{y}^n)$$

for any sequence $\sigma = (t_1, t_2, \ldots, t_n)$ of primitive terms of the form $t_i = t_i(\bar{y}^i)$ with the following properties: For any sequence $a = (\bar{a}^1, \ldots, \bar{a}^n)$ of sequences of types satisfying $\mathbf{Conds}(t_\sigma)$ which are indecomposable when they occur in positions corresponding to the $\mathbf{Vars}(t_\sigma)$,

$$\mathbf{TA} \vDash t[t_1[\bar{a}^1], \ldots, t_n[\bar{a}^n]] = t_\sigma[a].$$

Moreover, the following statements are equivalent for any two terms t, u.

(1) t = u holds identically in **TA.**

(2) For all σ, $t_\sigma = u_\sigma$ holds identically for the indecomposable types.

(3) 31-t-u.

We could extend our methods further to arbitrary terms of 3 with considerable complications in proofs. Theorem 7.3. is quite sufficient for our purposes.

PROOF: The first part follows from Lemma 7.6. and Theorem 7.1. By Lemma **7.3., (1) is** equivalent to (2). We need only show that (1) implies (3). Assume that (1) holds. By Theorem **7.2.,** each $t_\sigma = u_\sigma$ is provable in 3. By Lemma **7.6.,** we may assume that both t, u are *-primitive.

Let **TA'** be **the** subalgebra of **TA** generated by the indecomposable elements. Then any element $a \in$ **TA'** is a finite join of minimal elements. Let the ***support* of a; Supp(*a*); be the** (finite) set of all indecomposable types given via Lemma 7.3. Note the following fact: if ***F, G are*** two disjoint sets of indecomposable elements, then the intersection of the subalgebras **generated by** *F,* G respectively is the trivial subalgcbra generated by the empty set. This fact follows from the-first part of our theorem and Theorem 7.2.

We shall prove by induction on t the following fact for any finite set X of indecomposable elements: if $t \leq$ u holds for all elements of **TA'** satisfying $\mathbf{Conds}(t)$ whose support is disjoint **from X, then** 3 **t-t** $\leq$ **u.**

Without loss of generality, we may assume that t is *-minimal and u = $u_1 \vee \cdots \vee u_m$, where u_j are *-minimal. Since t is a minimal, it can be expressed as a product of atomic terms or terms in the form given by parts (1), (2) or (6) of Definition 7.5. We may assume that the product in question has only one element; i.e., t is one of the cases mentioned above. The case of products can be reduced to a system of "simultaneous" inequalities by taking projections. Our techniques apply with minor modifications in this case. We may also assume that t is not a constant.

Next part of the proof consists of showing that for some i the inequality t $\leq u_i$ holds for all elements of **TA'** satisfying **Conds**(t) whose support is disjoint from a set Y $\supseteq$ X. If not, we can construct type sequences $(\bar{a}_1, \ldots, \bar{a}_m)$ from **TA'** satisfying **Conds**(t) whose supports are mutually disjoint and disjoint from X such that for all i

$$t[\bar{a}_i] \not\leq u_i[\bar{a}_i].$$

Let $\bar{a}$ be the componentwise join of the $\bar{a}_i$. By the disjoint support assumption, we cannot have t $\leq$ u hold for this $\bar{a}$, a contradiction. Thus we may assume that u = u_1 is minimal; it is represented as a product. It is easy to see that this product cannot have more than one element.

It follows that both t, u must be terms in the form given by parts (1),(2) or (6) of Definition 7.5. If one of t, u is of the form x $\rightarrow$ y, then both must be and the result follows by induction. Othewise t must have the form $f_1(f_2(\cdots f_k(v)\cdots))$, $k \geq 0$ and u have the form $g_1(g_2(\cdots f_l(w)\cdots \bullet \quad))$, $l \geq 0$, where v, w are variables and f_i, g_j are projections such that only f_1 and g_1 can be a πl. By substituting nested products of indecomposable elements for v one can show that v = w, $k = l$ and $f_i = g_i$ for all i. Thus $\mathcal{T} \vdash t = u$.

8. Type Interpretations for Quantifier-free Terms

We can now express stable interpretations *as type interpretations.*

DEFINITION 8.1.: A type interpretation is a partial map f : A $\rightarrow$ **TA.** Each type interpretation

induces a partial map $\mathbf{Teval}(f) : \mathcal{T}_0 \to \mathbf{TA}$, as follows:

$\mathbf{Teval(f)(\ true)} = \mathbf{truthval},$

$\mathbf{Teval}(f)(\mathit{false}) = \mathbf{truthval},$

$\mathbf{Teval(f)(}\ \mathit{empty}) = \mathbf{empty},$

$\mathbf{Teval}(f)(t) = \mathbf{f(t)}, \quad \text{if } t \in \mathbf{A},$

$\mathbf{Teval}(f)(\mathit{tuple}(x_1, x_2, \ldots, x_n)) = \mathbf{Teval}(\ldots x_1) \otimes \ldots \otimes \mathbf{Teval}(\ )(x_n),$

$\mathbf{Teval}(f)(\pi_i(x_1, x_2, \ldots, x_n)) = \pi_i(\mathbf{Teval}(f)(x_1) \otimes \cdots \otimes \mathbf{Teval}(f)(x_n)), \quad i = 1, 2, \ldots,$

$\mathbf{Teval}(f)(\pi l_i(x_1, x_2, \ldots, x_n)) = \pi l_i(\mathbf{Teval}(f)(x_1) \otimes \cdots \otimes \mathbf{Teval}(f)(x_n)), \quad i = 1, 2, \ldots.$

The type interpretations of the operations A, $\vee, \neg, \supset$ and $\equiv$ follow standard rules:

$$\mathbf{Teval}(f)(x_1 \text{ A } x_2) = \begin{cases} \mathbf{truthval}, & \text{if } \mathbf{Teval}(f)(x_1) = \mathbf{Teval}(f)(x_2) = \mathbf{truthval}; \\ \perp, & \text{otherwise}, \end{cases}$$

$$\mathbf{Teval}(f)(x_1 = x_2) \begin{cases} = & \mathbf{truthval}, \quad \text{if } \mathbf{Teval}(f)(x_1) \neq \perp, \mathbf{Teval}(f)(x_2) \neq \perp; \\ & \perp, \quad \text{otherwise}, \end{cases}$$

$$\mathbf{Teval(f)(}\ \mathit{universal}(x_1)) = \begin{cases} \mathbf{truthval}, & \text{if } \mathbf{Teval}(f)(x_1) \neq \perp; \\ \perp, & \text{otherwise}, \end{cases}$$

$$\text{Teval(f)(if } x_1 \text{ then } x_2 \text{ else } x_3) = \begin{cases} \mathbf{Teval}(\text{f})(x_2) \vee \mathbf{Teval}(\text{f})(x_3), & \text{if } \mathbf{Teval}(f)(x_1) = \mathbf{truthval}; \\ \perp, & \text{otherwise}. \end{cases}$$

If h is not distinguished, and $\mathbf{Teval}(f)(\mathit{tuple}(x_1, x_2, \ldots, x_n)) \leq \mathbf{dom}\ \mathbf{Teval}(f)(h)$, then

$$\mathbf{Teval}(f)(h(x_1, x_2, \ldots, x_n)) = \mathbf{range}\ \mathbf{Teval}(f)(h).$$

DEFINITION 8.2.: A correct type *interpretation* for a term $t \in \mathcal{T}_0$ is a type interpretation $f : \text{A} \to \mathbf{TA}$ such that $\mathbf{Teval}(f)(t)$, the type of t under f, is $\neq \perp$. A term $t \in \mathcal{T}_0$ is ***well-typed*** if it has a correct type interpretation.

LEMMA 8.1.: If f is a correct type interpretation for $t \in \mathcal{T}_0$, then any type interpretation g such that for all x $\in$ Dom f, g(x) $\leq f$(x)is also a correct type interpretation for t. The type of t under g is $\leq$ the type of t under f.

THEOREM 8.1.: A term is well-typed iff it has stable interpretation.

PROOF: If a term t has a correct type interpretation f, then any g such that for all x $\in$ Dom f, g(x) $\in f(X)$ is a stable interpretation of t. The type of t under E(g) (see Definition 6.2) is $\leq$ to the type of t under f. The converse is equally easy to see.

THEOREM 8.2.: Let $t \in \mathcal{T}_0$ be a term with free variables $x_1, x_2, \ldots, x_n$. One can in a uniform way construct a formula $\phi = \phi(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y,)$, which is a conjunction of terms of the form $t_1 = t_2$ and a term τ of 3 with free variables $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$ not involving the operator * such that f is a correct type interpretation for t if and only if

$$\mathbf{TA} \models \exists y_1 y_2 \ldots y_m . \phi[f(x_1), \ldots, f(x_n), y_1, y_2, \ldots, y_m].$$

In this case,

$$\tau[f(x_1), \ldots, f(x_n), w_1, w_2, \ldots, w_n] = \mathbf{Teval}(f)(t)$$

for any sequence of types $(w_1, w_2, \ldots, w_n)$ satisfying the existential variables. Moreover, t has a type interpretation if and only if

$$\mathcal{F} \vdash \exists x_1 x_2 \ldots x_n y_1 y_2 \ldots y_m . \phi.$$

PROOF: ϕ is constructed inductively. Perhaps the most interesting case is that of application. Assume that $t = t_1(t_2)$ and we are given ϕ_i, τ_i for t_i. In this case, introduce a new variables y', y'' and let $\tau = y''$,

$$\phi = \phi_1 \wedge \phi_2 \wedge (\tau_2 = (Y' \rightarrow y'')) \wedge ((\tau_1 \vee y') = y').$$

The second part follows from Lemma **7.2. and Theorem 7.3.**

Finding a correct type interpretation for a term corresponds to a problem of proving an existential statement valid. By the results of section 7, this in turn can be reduced to **a set of** unification problems. For example, to type a formula of the form x A y we need to match it against the form **truthval A truthval.** In fact, this is precisely what EKL attempts to do when presented with a term with only partially known types. However, a "most general typing" **may** fail to exist for some terms. For example, the term $f(\pi_1(x))$ has no most general typing.

9. Extensions: Types, Sorts, Syntypes and Quantification

The notion of a ***context*** is of fundamental importance in the actual implementation of the logic of EKL. It is used to place further useful restrictions to the class of terms under consideration, both syntactically and semantically.

DEFINITION 9.1.: **A *context*** is a partial function C : A $\rightarrow \mathcal{V}$ such that for any atom x in the context, i.e., x $\in$ Dom C, C(x) is a triple consisting of the ***syntype, type*** and ***sort*** of x:'

$$C(x) = (\textit{Syntype-of}(x), \textit{Type-of}(x), \textit{Soft-of}(x)).$$

The syntype of an atom is a member of $\{constant, variable\}$. The type of an atom is a member of **TA.** The sort of an atom is a member of $\mathcal{T}$, which is assumed ***to*** be ***well-formed*** with respect to C; a term is well-formed with respect to C if it is built up from atoms in the context such that for any term of one of the forms $Make\text{-}exist(y, z), Make\text{-}universal(y, z), Make\text{-}lambda(y, z)$, where y is a tuple of atoms of ***variable*** syntype, z is a well-formed term, and all atoms in y have the sort ***universal in Make-lambda(y, z).***

From now on, all our discussion takes place relative to a fixed context **C.** In addition, we assign syntypes and sorts to the distinguished atoms in the most obvious way; they will all have the sort ***universal*** and ***constant*** syntype.

The function ***Type-of*** induces in a natural **way a type** interpretation for all quantifier-free well-formed terms.

DEFINITION 9.2.: The type of a term t, Type(t), is computed **as** follows: if t is quantifier-free, **then**

$$\mathbf{Type(t) = Teval(}\ \textit{Type-of)(t)}.$$

If t has the form $\forall x_1 x_2 \ldots x_n . u$ or $\exists x_1 x_2 \ldots$ x,. u, then

$$\mathbf{Type}(t) = \begin{cases} \mathbf{truthval}, & \text{if } \mathbf{Type}(u) = \mathbf{truthval}; \\ \perp, & \text{otherwise.} \end{cases}$$

If t has the form $\lambda x_1 x_2 \ldots x_n .$ u, then

$$\mathbf{Type(t)} = \mathbf{Type}(x_1) \otimes \ldots \otimes \mathbf{Type}(x_n) \rightarrow \mathbf{Type}(u).$$

A term t is ***well-typed*** if $\mathbf{Type(t)} \neq \perp$.

From now on, we shall assume that all terms are well-typed. In particular, **we assume that** for any atom x, the term ***(Sort-of (x))(x)*** is well-typed and of type **truthval.**

DEFINITION 9.3.: **A** function $f \in \mathbf{Env}$ is an environment relative to the context C if for all $x \in$ A,

(1) If x has type ***A,*** then $f(x) \in A$.

(2) If x has sort ***a,*** then $(\mathbf{Eval}(f)(a))(f(x)) = \mathbf{true}$.

A context ***is con&tent*** if there is at least one environment relative to it.

We shall assume that C is consistent. We may think of a type as a syntactic restriction and a sort as a semantic restriction. The function **Eval** is extended in **a** natural way to handle terms with quantifiers.

DEFINITION 9.4.: Given a term t and atoms $x_1, x_2, \ldots, x_n$ of ***variable*** **syntype define a partial** function $\mathbf{Eval}(f; x_1, x_2, \ldots, x_n)(t) : \mathcal{V}^n \to \mathcal{V}$ as follows. For any $h \in \mathcal{V}^n$ such that the **function** g defined by $\mathbf{Dom}(g) = \{x_1, x_2, \ldots, x_n\}, g(x_i) = h(i)$ is an environment relative to C, set

$$[\mathbf{Eval}(f; x_1, x_2, \ldots, x_n)(t)](h) = \mathbf{Eval}(g\#f)(t).$$

If t has the form $\forall x_1 x_2 \ldots x_n$. **u and** R = $\mathbf{Range}(\mathbf{Eval}(f; x_1, x_2, \ldots, x_n)(u)) \subseteq \mathbf{Boole}$ **then**

$$\mathbf{Eval}(f)(t) = \begin{cases} \mathbf{true}, & \text{if false} \notin R; \\ \mathbf{false}, & \text{otherwise.} \end{cases}$$

If t has the form $\exists x_1 x_2 \ldots x_n$. u and R = $\mathbf{Range}(\mathbf{Eval}(f; x_1, x_2, \ldots, x_n)(u)) \subseteq \mathbf{Boole}$ **then**

$$\mathbf{Eval}(f)(t) = \begin{cases} \mathbf{true}, & \text{if true} \in R; \\ \mathbf{false}, & \text{otherwise.} \end{cases}$$

If t has the form $\lambda x_1 x_2 \ldots x_n . u$, then

$$\mathbf{Eval}(f)(t) = \mathbf{Eval}(f; x_1, x_2, \ldots, x_n)(u).$$

From now on, all environments are given relative to the context **C.**

It is worthwhile to define few more operations on $\mathcal{T}$; namely, substitution for atoms and computation of free variables.

DEFINITION 9.5.: A ***substitution list*** is a partial function $f : A \rightharpoonup \mathcal{T}$ such that for any x, $\mathbf{Type}(f(x)) \leq$ Type(z). Each substitution list induces a partial function $Subst(f) : \mathcal{T} \rightarrow \mathcal{T}$. The functions $Subst(f)$ and $Freevars : \mathcal{T}^* \rightarrow V$ are defined by induction as follows. If x is an atom, set

$$Subst(f)(x) = \begin{cases} f(x), & \text{if } x \in \mathbf{Dom}\ f; \\ x, & \text{otherwise,} \end{cases}$$

and

$$Freevars(x) = \begin{cases} \{x\}, & \textit{if Syntype-of(x)} = \textit{variable;} \\ \emptyset, & \text{otherwise.} \end{cases}$$

Freevars and *Subst* are extended to $\mathcal{T}^*$ inductively:

$$Freevars(x_1, x_2, \ldots, x_n) = \bigcup\{Freevars(x_1), \ldots, Freevars(x_n)\},$$

$$Subst(f)(x_1, x_2, \ldots x_n) . (Subst(f)(x_1), \ldots, Subst(f)(x_n)),$$

$$Freevars(Make\text{-}application(x, y)) = Freevars(x) \cup Freevars(y),$$

$$Subst(f)(Make\text{-}application(x, y)) = Make\text{-}application(Subst(\ j)(x), Subst(\ f)(y)),$$

$$Freevars(Make\text{-}exist(y, z)) = Freevars(z) - Freevars(y),$$

$$Freevars(Make\text{-}universal(y, z)) = Freevars(z) - Freevars(y),$$

$$Freevars(Make\text{-}lambda(y, z)) = Freevars(z) - Freevars(y),$$

$$Subst(f)(Make\text{-}exist(y, z)) = Make\text{-}exist(y, Subst(f)(z)),$$

$$Subst(f)(Make\text{-}universal(y, z)) = Make\text{-}universal(y, Subst(f)(z)),$$

$$Subst(f)(Make\text{-}lambda(y, z)) = Make\text{-}lambda(y, Sub@)(a)).$$

In order to prevent captures of bound variables, we require that in the last three equations $y \cap \mathrm{Dom}\ f = \emptyset$, and for all v $\in$ Dom f, $y \cap Freevars(f(v)) = \emptyset$. In the actual implementation of EKL, these restrictions are circumvented by renaming bound variables.

DEFINITION 9.6.: A variable x ***occurs free*** in the term t if $x \in Freevars(t)$.

LEMMA 9.7.: If f is a subsitution list, then for any well-typed term t, the term $Subst(f)(t)$ *is* well- typed.

Given a substitution list f such that Dom $f = \{x_1, x_2, \ldots, x_n\}$, $f(x_i) = t_i$, we use the notation $u[x_1/t_1, \ldots, x_n/t_n]$ for $Subst(f)(u)$.

10. The Logic of EKL: Axioms, Validity and Soundness

DEFINITION 10.1.: A term t is valid if for all environments f, $\mathbf{Eval}(f)(t) = \mathbf{true}$.

It is easy to see that any axiom of propositional logic is valid. Similarly for axioms of predicate logic, with suitable modifications:

LEMMA 10.2.: If x is a variable of sort S, then following formulas are valid, when suitably typed:

$$A[x/t] \wedge S(t) \supset \exists x.A,$$
$$\forall x.\ A \supset (S(t) \supset A[x/t]).$$

Lemma 10.2. is an immedediate consequence of the *substitution property* of our semantics.

LEMMA 10.3.: If x is a variable, t a term, h an environment such that $\mathbf{Eval}(h)(t) = \mathbf{v}$ and f is an environment such that Dom f = {x} and $f(x) = \mathrm{v}$, then for any term A

$$\mathbf{Eval(f\#h)(A)} = \mathbf{Eval}(h)(A[x/t]).$$

We can prove that the standard rules of inference, modeled after [Kleene 1952] are valid, i.e., application of these rules yields valid conclusions from valid premises.

LEMMA 10.4.: The following two rules are valid, where v is a variable not occurring free in C and x has sort S:

$$\frac{C \supset (S(v) \supset A)}{C \supset \forall x.A[v/x]}, \qquad \frac{S(v) \wedge A \supset C}{\exists x.\ A[v/x] \supset C}.$$

We also have a rule of replacement:

LEMMA 10.5.: Assume t and u are terms such that $\mathbf{Type(t)} \leq \mathbf{Type(u)}$. Then the following rule is valid:

$$\frac{t = u, A}{A[u/t].}$$

Note that the type restrictions for equality are rather loose. In particular, the transitivity rule for equality is not valid. Aside from this, one can show that most, other standard facts from logic carry through without modification. There are other, slightly more interesting facts for EKL:

THEOREM 10.1.: The following terms are valid, when suitably typed:

$universal(x)$,

$S(x)$, if x has sort S,

$(x) = x$,

$\pi_i(\pi l_j(x)) = \pi_{i+j}(x)$,

$\pi l_i(\pi l_j(x)) = \pi l_{i+j}(x)$,

$x_n \neq () \supset \pi_i(x_1, x_2, \ldots, x_n) = x_i$, if $1 < n, i < n$,

$x_n \neq () \supset \pi_i(x_1, x_2, \ldots, x_n) = \pi_{n-i+1}(x_n)$, if $1 < n, i \geq n$,

$x_n \neq () \supset \pi l_i(x_1, x_2, \ldots, x_n) = (x_{i+1}, \ldots, x_n)$, if $1 < n, i < n$,

$x_n \neq () \supset \pi l_i(x_1, x_2, \ldots, x_n) = \pi l_{n-i+1}(x_n)$, if $1 < n, i \geq n$,

$(x_1, x_2, \ldots, x_n, \ldots, (y_1, \ldots, y_m)) = (x_1, x_2, \ldots, x_n, \ldots, y_1, \ldots, y_m)$,

$(x_1, x_2, \ldots, x_n, ()) = (x_1, x_2, \ldots, x_n)$,

$P(\text{if } a \text{ then } b \text{ else-c}) = \text{if } a \text{ then } P(b) \text{ else } P(c)$,

(if a then b else c)(x) = if a then $b(x)$ else c(x),

$(\lambda x_1 x_2 \ldots x_n . P)(t_1, t_2, \ldots t_n) = P[x_1/t_1, \cdots, x_n/t_n]$.

If the type of x contains no product types,

$\pi_i(x) = x, \qquad \pi l_i(x) = x.$

If x and y have the same type, x has sort S and y has sort ***universal,***

$\exists x.\ P(x) \equiv \exists y.\ S(y)\ A\ P(y), \qquad \forall x.\ P(x) \equiv \forall y.\ S(y) \supset P(g).$

The statements given in Theorem 10.1. can be viewed as axioms for the logic of EKL along with the rules of inference stated above. We could then consider EKL to be a variant of ordinary high-order logic satisfying many of the nice proof-theoretic properties (deduction theorem, normalization, etc.) of the more traditional **logics** given, say, by [Prawitz **1965**].

We can also formulate the principle of ***tuple induction:***

THEOREM 10.2.: If x is a variable of type $\mathbf{t^*}$ and y is a variable of type $\mathbf{t}$ and universal sort and P is a variable of type $\mathbf{t^* \rightarrow truthval}$, then

$$\forall P.P() \ A\ (\forall x\ y.\ P(x) \supset P(y, x)) \ 3\ \forall x.P(x).$$

Given that we have natural semantics for this system, it immediately follows that it is both ***sound*** and ***consistent.***

11. Further extensions: Absolute constants

We postulate a "quote" operator for EKL — this is regarded as an **injective** partial function '$: V \times TA \to A$ such that for any $x \in V$, type A with $x \in A$, $('x, A)$ is an atom of EKL of type A, sort ***universal*** and **syntype *constant.*** When the type of $'(x, A)$ is well understood, $'x$ is used to denote this atom. We may also use the notation $\uparrow t$ for $'t$ when t is a well-formed term. The quote map gives us a way of talking about objects in the "real" world, including terms of our language itself, regarded as set-theoretic objects.

The semantics given above can be easily extended to absolute constants. For any $f \in$ **Env,**

$$\mathbf{Eval}(f)('x) = x.$$

LEMMA 11.1.: For any function f, sets $x_1, x_2, \ldots, x_n, y$ such that $f(x_1, x_2, \ldots, x_n) = y$, the following formula is valid, when properly typed:

$$'f('x_1, 'x_2, \ldots, 'x_n) = 'y.$$

Similarly,

$$true = 'true, \qquad false = 'false.$$

Note that for any function f, EKL contains the axiom schema given above--thus the language of EKL becomes highly non-recursive. The object f is an extensional **entitity;** it is a **"real"** function as opposed to a term or a program computing a function. In the **actual** implementation of EKL we have *a* fixed list, of quoted objects, including numerals and functions like +,* and ***Make-application.*** Verifying the correctness of statements like **"5+9=14"** is then a simple matter of computation, instead of deduction from first, principles.

Lemma 11.1. legitimizes in our system the notion of ***semantic attachment*** that was first proposed by [Weyhrauch **1978**] in a more intensional context. In spirit it is, however, closer to the theory proposed by [Boyer-Moore **1979B**].

12. Meta Theory: Formal evaluation

We have constructed the quote operation in rough analogy with the LISP programming language. While for obvious reasons we one cannot say too much about the properties of "quote,,, it would be desirable to have an "inverse,'operation, corresponding to evaluation of LISP S-expressions. In our context, the "formal evaluation **operator"** will be denoted by **"↓"** — a new object to be added **to** the list ***D*** of distinguished symbols. Great care must be taken in assigning semantics to ↓; we need to avoid complications in inference rules and at the same time unintended captures in binding contexts. Consider, for example, the following fallacious chain of inference:

$x = \downarrow\uparrow x$	the value of a quoted term is . . .
vx. $x = \downarrow\uparrow x$	universally generalize . . .
$1 = \downarrow\uparrow x$	specialize the only variable x under **V to 1 . . .**
1 = **x**	**ahah!**

To solve these problems, each term of the form $\downarrow s$ is coupled with an environment. The typing rules and semantics can then be defined as follows:

DEFINITION 12.1.: For any term ***s*** whose type is a set of terms and any environment, ***h*** for these terms, $\downarrow[s, h]$ **is** a term such **that**

$$\mathbf{Type}(\downarrow[s, h]) = \bigcup\{\mathbf{Type}(x) : x \in \mathbf{Type}(s)\},$$

and for any environment, ***f***

$$\mathbf{Eval}(f)(\downarrow[s,h]) = \mathbf{Eval}(h)(\mathbf{Eval}(f)(s)).$$

LEMMA 12.1.: The substitution property of Lemma 10.3 still holds for the extended language.

PROOF: We have to verify by induction on the term s that for all variables x, terms t, environments h the following fact holds: **if** $\mathbf{Eval}(h)(t) =$ v and f is an environment such that Dom $f =$ {x} and j(x) = v, then

$$\mathbf{Eval}(f \# h)(s) = \mathbf{Eval}(h)(s[x/t]).$$

The only interesting situation occurs when s has the form $\downarrow[u,$ e] for some environment e. In that **case,**

$$\mathbf{Eval}(f \# h)(s) = \mathbf{Eval}(e)(\mathbf{Eval}(f \# h)(u)),$$

and

$$\mathbf{Eval}(h)(s[x/t]) = \mathbf{Eval}(e)(\mathbf{Eval}(h)(u[x/t])),$$

so the claim follows by induction again.

One can prove again that the soundness and the consistency of the formal system described before **is preserved.**

The formal properties of expressions of the form $\downarrow\uparrow x$ are somewhat different from those implied by our example.

THEOREM 12.1.: If t is any term, then the following formula is valid:

$$\downarrow[\uparrow t, h] = '(\mathbf{Eval}(h)(t)).$$

In particular, if t has no free variables, then

$$\downarrow\uparrow t = t.$$

PROOF: By definition, for any environment f,

$$\mathbf{Eval}(f)(\downarrow[\uparrow t, h]) = \mathbf{Eval}(h)(t).$$

If t has no free variables, then for any f

$$\mathbf{Eval}(f)(t) = \mathbf{Eval}(h)(t) = \mathbf{Eval}(f)(t)['(\mathbf{Eval}(h)(t))].$$

Theorem 12.1. shows that ↓ does indeed formalize the properties of **Eval** in V. If we choose not to introduce new quoted terms, we can express Theorem 12.1. as follows:

THEOREM 12.2.: If t is an arbitrary term, and t' is the term obtained by replacing each free variable x in t by an expression of the form $\downarrow[\uparrow x, h]$, then

$$\downarrow[\uparrow t, h] = t'.$$

PROOF: It is easy to prove by induction on t that for any environments e, f,

$$\mathbf{Eval}(e\#h)(t) = \mathbf{Eval}(e\#f)(t''),$$

where t'' is obtained by replacing each free variable x in t which does not occur in Dom e by an expression of the form $\downarrow[\uparrow x, h]$. Thus, for any environment f

$$\mathbf{Eval}(f)(\downarrow[\uparrow t, h]) = \mathbf{Eval}(h)(t) = \mathbf{Eval}(f)(t').$$

In fact, we can *axiomatize the semantics* for quantifier-free terms in the current framework. For example, the following statement is valid: for suitably typed variables x and y,

$$\downarrow['\mathit{Make\text{-}application}(x,y), h] = \mathit{Make\text{-}application}(\downarrow[x, h], \downarrow[y, h]).$$

For quantified terms such commutativity rules are harder to state. For example,

$$\downarrow['\mathit{Make\text{-}lambda}('(x), \uparrow(y(x))), h] = '(\mathbf{Eval}(h)(y)),$$

whereas $\downarrow[\uparrow(y(x)), h]$ may not make sense.

For validity, we can also state a trivial "reflection principle:,,

$$\forall s.s = \mathit{'true} \supset \downarrow s$$

13. Examples of EKL expressions

The use of propositional types

EKL makes no distinction between terms and formulas. Formulas are simply terms of type **truthval.** For instance, the statement

$$\forall x\ y.\ \neg(x \vee y) \equiv (\neg x) \wedge (\neg y)$$

expresses a familiar law of De Morgan. Type restrictions prevent us from forming such obviously contradictory expressions as Xx. $\neg x(x)$.

The use of high-order type8 and list types

It seems that most of the concepts regarded as "metatheoretic" can be naturally expressed in terms of high-order predicate logic. The need for **meta-theory** has in many cases been an artifact of restriction into first-order expressions. For example, facts about simple schemata can be formulated in terms of second-order quantifiers; the induction **schema**

$$\phi(0) \wedge (\forall n.\ \phi(n) \supset \phi(n + 1)) \supset \forall n.\ \phi(n)$$

can be equally well expressed as the second-order sentence

$$\forall P.\ P(0) \wedge (\forall n.\ P(n) \supset P(n + 1)) \supset \forall n.\ P(n).$$

The use of list types gives a way of talking about parameterized schemata. For example, we can express the traditional function definition by primitive recursion given by [Kleene **1952**], page 219, as the following sentence:

$$\forall initfun\ indcase.\ \exists fun.\ \forall pars.\ fun(0) = initfun(pars) \wedge \forall n.\ fun(n + 1, pare) = indcase(n, fun(n, pars), pars)'$$

Here *par8* is a variable of list type; it, can match to any sequence of (suitably **typed**) variables, including the null sequence. This sentence can then be automatically instantiated to **a** desired definition by the use of ***high order unification;*** see [Huet **1975**]. Examples of the use of the EKL unification mechanism are given in **[Ketonen 1983].**

The use of list types allow us **to** talk about functions of arbitrary number of variables in **a** natural way. Consider, for example, the problem of expressing the following statement in our

formalism:

$$x \in \{x_1, x_2, \ldots, x_n\} \equiv (x_1 = x) \vee \cdots \vee (x_n = x).$$

This can be interpreted **as a schema:**

$$\text{Vs. x} \in \{\} \equiv \text{false},$$
$$\forall x\, y.\, x \in \{y\} \equiv (x = y),$$
$$\forall x\, y\, z.\, x \in \{y, z\} \equiv (x = y \vee x = z),$$
$$\ldots$$

leaving it up to a **meta-theoretic** apparatus to formulate the obvious inductive function that constructs these sentences. The full formulation in this manner would be quite painful. Another alternative is to use list induction to achieve the same result. Here we show how this is actually done **in EKL. The LISP** function DECL declares new atoms; TRW expands definitions by rewriting,

```
;first some declaration8
(DECL SET (TYPE: I GROUND*|))
(DECL ELEMENT (TYPE: |GROUND|))
(DECL ELEMENT_EQUAL (TYPE: |GROUND⊗(GROUND*)→TRUTHVAL|) (SYNTYPE: CONSTANT))
(DECL MEMBER (TYPE: |GROUND⊗GROUND→TRUTHVAL|) (INFIXNAME: ∈) (BINDINGPOWER: 925))
;definition for X equal to a member of SET
(AXIOM |∀X Y.ELEMENT_EQUAL(X,())=FALSE|)
(LABEL EQUAL)
(AXIOM |∀X Y SET.ELEMENT_EQUAL(X,Y,SET)=((X=Y)∨ELEMENT_EQUAL(X,SET))|)
(LABEL EQUAL)
;definition of membership
(AXIOM |∀X SET.X∈{SET}=ELEMENT_EQUAL(X,SET)|)
(LABEL MEMBERDEF)
;expanding the definition...
(TRW |X∈{}| (USE (MEMBERDEF EQUAL) MODE: ALWAYS))
;¬X∈{}
;and so on
(TRW |X∈{Y,Z}| (USE (MEMBERDEF EQUAL) MODE: ALWAYS))
```

```
;X∈{Y,Z}≡X=Y∨X=Z
```

The use of meta theory

Suppose one has a decision procedure $D : \mathcal{T} \rightarrow \{0,1\}$ such that if $D(t) = 0$, then t is a valid sentence. Then one may want to state this property of D as an EKL expression:

$$\forall s.'D(s) = 0 \supset \downarrow s.$$

References

[Boyer and Moore 1979A]

Boyer, R. S., Moore, J. S., **A Computational Logic,** Academic Press, New York, 1979.

[Boyer and Moore 1979B]

Boyer, R. S., Moore, J. S., *Metafunctions:* Proving *them correct* and *using* them *efficiently* as new proof procedures, SRI International, Technical Report CSL-108, 1979.

[de Bruijn 1968]

de Bruijn, *N.G., AUTOMATH—A* Language *for* Mathematics, Technological University Eindhoven, Netherlands, 1968.

[Feferman 1977]

Feferman, S., *Theories of* Finite Type *Related to* Mathematical Practice, in "Handbook of Mathematical Logic," edited by J. Barwise, 913-971, North Holland, 1977.

[Goguen 1978]

Goguen, J. A., Abstract Errors for Abstract Data Types, in "Formal Descriptions of Programming Concepts", edited by E. J. Neuhold, 491-525, North Holland, 1978.

[Huet 1975]

Huet, G. P., A Unification Algorithm for *Typed λ-Calculus*, Theoretical Computer Science 1, 27-57, 1975.

[Ketonen 1983]

Ketonen, J., *EKL—A* Mathematically Oriented Proof *Checker*, 16 pp., Stanford University, 1983.

[Ketonen and Weening 1983]

Ketonen, J., Weening, J. S., *EKL—An Interactive Proof* Checker, Users' Reference Manual, 40 pp., Stanford University, 1983.

[Kleene 1952]

Kleene, S. C., **Introduction to. Metamathematics,** Van Nostrand, 1952.

[Prawitz 1965]

Prawitz, D., **Natural Deduction: A Proof-theoretical Study, Almquist and Wiksell, 1965.**

[Scott 1980]

Scott, D., Relating Theories *of* the X-calculus, in "To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism", edited by J. P. Seldin and J. R. Hindley, 403-450, Academic Press, 1980.

[Weyhrauch 1978]

Weyhrauch, R. *Prolegomena* to a *theory of mechanized* formal reasoning, Stanford AI Memo AIM-315, 1978.