Theoretical Computer Science 22 (1983) 19–55
North-Holland Publishing Company

THEORY OF SYMBOLIC EXPRESSIONS, I

Masahiko SATO
Department of Information Science, University of Tokyo, Bunkyo-Ku, Tokyo, Japan

Communicated by M. Nivat
Received July 1981

Introduction

The purpose of this series of papers is to introduce a new domain **S** of symbolic expressions (*sexps*, for short), and to study finite mathematics within the framework of **S**. Although finite mathematics contains traditional mathematical theory such as number theory or finite set theory, our emphasis will be mainly on the theory of computation including its metatheory.

Finite mathematics deals with *finitary objects* such as natural numbers, strings of symbols or proof trees. Our domain **S** is flexible enough to permit natural *representation* of these finitary objects. Thus, for instance, a natural number n can be represented by a certain symbolic expression s in **S**. Similarly a proof tree T may be represented by a sexp t. We will, however, consider s not as a representation of a natural number but as the natural number n itself. Similarly we will consider t as a proof tree. It may well be the case that $s = t$. In such a case, it is our *viewpoint* that determines whether it is (considered to be) a natural number or a proof tree. In our theory, every finitary object *is* a sexp. (Note that a similar standpoint is taken in set theory, where every object is a set.) In this way, our domain **S** becomes a universal domain of finitary objects. The principle that every finitary object is a sexp may seem quite innocent, but it will have important consequences throughout our study.

An intuitive definition of a sexp can be given as follows. Imagine an infinite leaf-free binary tree like Fig. 1, where a small circle is drawn at each node. The topmost node is called the *root*. Choose a finite number of nodes arbitrarily and mark them black as in Fig. 2. We call the resulting figure a *sexp*. The sexp with no marked nodes is denoted by 0. Fig. 1, considered as a sexp, is 0. The sexp whose only marked node is the root is denoted by 1. A sexp is called an *atom* if its root is marked. A sexp whose root is unmarked is called a *molecule*. For any sexp z, its left subtree is called the *car* of z and its right subtree is called the *cdr* of z. For instance, the car and the cdr of 0 and 1 is 0. For any sexp x and y, there uniquely exists a molecule z whose car is x and whose cdr is y. We call such a z the *cons*

0304-3975/83/0000–0000/$03.00

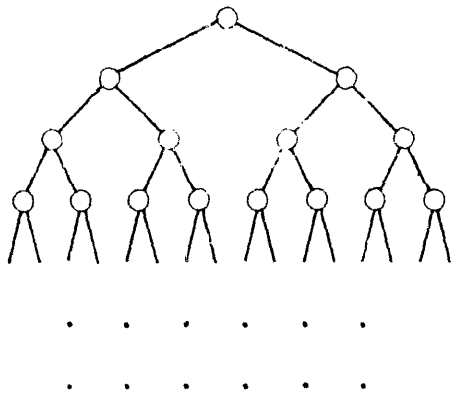

Fig. 1.

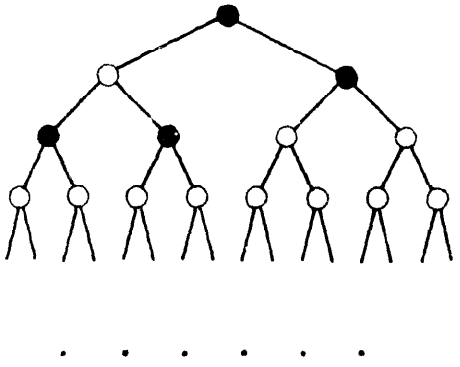

Fig. 2.

of x and y. If we mark the root of z, we obtain the *snoc* of x and y. For instance, the cons of 0 and 0 is 0 and the snoc of 0 and 0 is 1. We remark that every sexp can be constructed from 0 by a finite number of applications of cons and snoc. The resulting domain **S**, which consists of all the sexps, is mathematically much neater than the classical domain of Lisp symbolic expressions.

As a first step towards a theory of symbolic expressions, we introduce in Section 5 a programming language called Hyperlisp. In accordance with the principle stated above, a Hyperlisp program – which is a finitary object – is just a sexp. Thus, taking account of the possibilities of nontermination of evaluation or erroneous termination, the semantics of Hyperlisp will be given by a (recursively enumerable) binary relation $eval \subset \mathbf{S} \times \mathbf{S}$ such that $eval(x, y)$ and $eval(x, z)$ implies $y = z$ (i.e., *eval* is a partial map: $\mathbf{S} \to \mathbf{S}$). The intuitive meaning of $eval(x, z)$ is that x is evaluated to z. In order to define the set *eval* formally, we introduce the concept of a *formal system* in Section 2.

In accordance with the principle stated above, a formal system is defined as a sexp, and we can define any recursively enumerable subset of **S** in terms of a formal system. The definition of a recursively enumerable set of sexps given in Section 2 is equivalent to the usual definition of a r.e. set of natural numbers, if we identify the set **S** with the set **N** of natural numbers through the following bijective map. The sexp 0 is mapped to the natural number 0. If x, y are mapped to m, n respectively, then the cons of x and y is mapped to $k = (m+n)(m+n+1)+2m$ and the snoc of x and y is mapped to $k+1$. Since cons and snoc enjoy the property of a pairing function, Cartesian products of (subsets of) **S** will be defined as subsets of **S**. The set *eval* then becomes a r.e. subset of **S** and we can define *eval* in terms of a formal system.

A convenient notation system for sexps is necessary in order to express Hyperlisp programs succinctly. The *reference language* will be introduced in Section 4 to provide such a notation system. The reference language is also defined formally in terms of a formal system **Ref**.

Since our definition of the semantics of Hyperlisp is formal, it is possible to rigorously state and prove the *correctness* of a (metacircular) interpreter of Hyperlisp. This is one of the main result of this paper, and is established in Section 5. Another basic result of this paper is a proof, without employing any Gödel numbering or coding, of the well-known theorem which asserts the existence of a recursively enumerable but nonrecursive subset of S. This result is proved in Section 3.

The first interpreter of Hyperlisp was implemented by Masami Hagiya of the University of Tokyo on a PDP-11 under UNIX.

An informal but precise introduction to Hyperlisp can be found in Sato and Hagiya [10]. It also explains some features of Hyperlisp not discussed in this paper.

We also remark that the domain S can be made into a noncommutative ring by defining addition and multiplication suitably. The sexps 0 and 1 respectively become the unit element of addition and multiplication. The study of such algebraic structure of S is, however, out of the scope of the present paper. For this, we refer the interested reader to Sato [9]

1. Symbolic expressions

In this section we define symbolic expressions as concrete finitary objects. Let the *basic language* consist of:

(1) function symbols:

0 (nullary)

a, **d** (unary)

c, **s** (binary)

(2) variables:

$x, y, z, \ldots$

Remark. **a**, **d**, **c** and **s** stand for car, cdr, cons and snoc, respectively.

We define *basic terms* inductively as follows. We will use lower case Greek letters as syntactic variables.

(1) **0** is a basic term.

(2) If α is a variable, then it is a basic term.

(3) If α is a basic term, then so are $\mathbf{a}\alpha$ and $\mathbf{d}\alpha$.

(4) If α, β are basic terms, then so are $\mathbf{c}\alpha\beta$ and $\mathbf{s}\alpha\beta$.

A basic term is *closed* if it does not contain variables. A basic term is *normal* if (i) it is constructed only by using the clauses (1) and (4) and (ii) it does not contain occurrences of **c00**. A normal basic term is also called a *symbolic expression* (*sexp*, for short). We will use ρ, σ, τ etc. to denote sexps. Note that any sexp is one of the following forms:

(i) **0**

(ii) $\mathbf{c}\sigma\tau$ where σ, τ are sexps and at least one of them is not equal to **0**,

(iii) $\mathbf{s}\sigma\tau$ where σ, τ are sexps.

We denote the set of all the sexps by **S**. A sexp of the form (i) or (ii) is called a *molecule*, and a sexp of the form (iii) is called an *atom*. We denote the set of all the molecules (atoms) by **M** (**A**, resp.). Let **V** denote the set of all the variables. A mapping $\Delta: \mathbf{V} \to \mathbf{S}$ is called an *environment* if the set of variables whose image under Δ is not equal to **0**, is a finite set. Relative to an environment Δ, we define a relation $\blacktriangleright$ between basic terms and sexps, inductively as follows. We will say that α *denotes* σ if $\alpha \blacktriangleright \sigma$.

(1) $\mathbf{0} \blacktriangleright \mathbf{0}$,

(2) If $\chi \in \mathbf{V}$ then $\chi \blacktriangleright \Delta\chi$,

(3.1) If $\alpha \blacktriangleright \mathbf{0}$ then $\mathbf{a}\alpha \blacktriangleright \mathbf{0}$ and $\mathbf{d}\alpha \blacktriangleright \mathbf{0}$

(3.2) If $\alpha \blacktriangleright \mathbf{c}\sigma\tau$ then $\mathbf{a}\alpha \blacktriangleright \sigma$ and $\mathbf{d}\alpha \blacktriangleright \tau$,

(3.3) If $\alpha \blacktriangleright \mathbf{s}\sigma\tau$ then $\mathbf{a}\alpha \blacktriangleright \sigma$ and $\mathbf{d}\alpha \blacktriangleright \tau$,

(4.1) If $\alpha \blacktriangleright \mathbf{0}$ and $\beta \blacktriangleright \mathbf{0}$ then $\mathbf{c}\alpha\beta \blacktriangleright \mathbf{0}$,

(4.2) If $\alpha \blacktriangleright \sigma$ and $\beta \blacktriangleright \tau$ and σ or τ is not equal to **0** then $\mathbf{c}\alpha\beta \blacktriangleright \mathbf{c}\sigma\tau$

(4.3) If $\alpha \blacktriangleright \sigma$ and $\beta \blacktriangleright \tau$ then $\mathbf{s}\alpha\beta \blacktriangleright \mathbf{s}\sigma\tau$

We write $\alpha \blacktriangleright \rho$ (under Δ) if we wish to emphasize the dependence on Δ. Note that any basic term denotes a unique sexp under a fixed environment. Two terms are *equal* (=) if they denote the same sexp. We use the symbol $\simeq$ to denote *syntactic identity* of terms. Thus, e.g., $\mathbf{c00} = \mathbf{0}$ is true but $\mathbf{c00} \simeq \mathbf{0}$ is false. We summarize basic properties of the domain **S** in the following theorem. We will omit the proof of easily verifiable propositions. This is the first example of such propositions.

Theorem 1.1 (i) $\mathbf{ac}xy = \mathbf{as}xy = x$.

(ii) $\mathbf{dc}xy = \mathbf{ds}xy = y$.

(iii) $\mathbf{c00} = \mathbf{0}$.

(iv) $\mathbf{c}xy \neq \mathbf{s}uv$.

(v) *Let* $\Phi[x]$ *be a proposition about a sexp* x.
If $\Phi[\mathbf{0}]$ *is true and*
$\Phi[x]$ *and* $\Phi[y]$ *implies* $\Phi[\mathbf{c}xy]$ *and* $\Phi[\mathbf{s}xy]$
then $\Phi[x]$ *holds for all* x

In order to provide a more convenient notation for sexps, we introduce the *mini language*. The alphabet $\mathbf{L}_{\text{mini}}$ of the mini language consists of:

(1) **lowercase:**

a b c d e f g h i j k l m n o p q r s t u v w x y z

(2) **uppercase:**

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(3) **digit:**

0 1 2 3 4 5 6 7 8 9

(4) **special characters:**

( ) [ ] , ; . →

(5) **variables:**

$x\ y\ z \cdots X\ Y\ Z \cdots$

(6) **function-symbols:** they are introduced as necessary.

lowercase, uppercase and **digit** are collectively called **alphanumeric**. By a **literal (metaliteral)** we mean a finite nonempty sequence of **alphanumeric** characters whose first character is a **lowercase (uppercase,** resp.). We use (a variant of) BNF notation to define the syntax of the mini language. (A sequence of bold face characters such as **term, unit** etc. represents a syntactic category.)

term ::= **unit**|**sequent**
sequent ::= **const** ; . . . ; **const** → **const**.
unit ::= **unit list**|**function-symbol list**|**literal**|**metaliteral**|
variable|**list**|0|1
const ::= *a* **unit** *containing no* **function-symbol**'*s*.
list ::= **conslist**|**snoclist**
conslist ::= (**term**, . . . , **term** . **term**)|(**term**, . . . , **term**)
snoclist ::= [**term**, . . . , **term** . **term**]|[**term**, . . . , **term**]

where **const**; . . . ; **const** (**term**, . . . , **term**) means a possibly empty sequence of **const**'s (**term**'s) separated by semicolons (commas, resp.). With each metaliteral α we associate a literal $\underline{\alpha}$ which is obtained from α by replacing its first character by the corresponding **lowercase** character. For instance, if α is Goto then $\underline{\alpha}$ is goto. Also, with each **alphanumeric** character α, we associate a **list** $\tilde{\alpha}$ as follows. First we observe that each **alphanumeric** is an ASCII character and hence has the corresponding 7 bit ASCII code. For instance, if $\alpha = Z$ the ASCII code of α is 1011010 in binary. We represent this code by the **list** [1, 0, 1, 1, 0, 1, 0] and we define $\tilde{\alpha}$ as this **list**.

Now, relative to an environment Δ, we define a relation ▶ between **term**'s and sexps inductively as follows. At the same time, a relation > between **alphanumeric-seq** (i.e., a possibly empty sequence of **alphanumeric**) and sexps is also defined as an auxiliary relation. We will say that α *denotes* σ (or, σ is the *denotation* of α) if $\alpha \blacktriangleright \sigma$. In the following definition, **empty** denotes empty sequence:

(1.1) $0 \blacktriangleright$ **0**.
(1.2) $1 \blacktriangleright$ **s00**.
(1.3) χ: **variable** $\Rightarrow \chi \blacktriangleright \Delta\chi$.
(1.4) α: **literal** $\alpha > \sigma \Rightarrow \alpha \blacktriangleright \sigma$.
(1.5) α: **metaliteral**, $\underline{\alpha} > \sigma \Rightarrow \alpha \blacktriangleright \sigma$.
(1.6) α: **term** $\blacktriangleright \sigma \Rightarrow (\cdot\ \alpha) \blacktriangleright \sigma$.
(1.7) $\alpha_1, \ldots, \alpha_n$: **term** $(n \geq 1)$, β: **term**, $\alpha_1 \blacktriangleright \sigma$,
$(\alpha_2, \ldots, \alpha_n \cdot \beta) \blacktriangleright \tau \Rightarrow (\alpha_1, \ldots, \alpha_n \cdot \beta) \blacktriangleright$ **c**$\sigma\tau$.

(1.8) $\alpha_1, \ldots, \alpha_n$: **term** $(n \geqslant 0)$, $(\alpha_1, \ldots, \alpha_n \cdot 0) \blacktriangleright \sigma$
$\Rightarrow (\alpha_1, \ldots, \alpha_n) \blacktriangleright \sigma$.

(1.9) α : **term** $\blacktriangleright \sigma \Rightarrow [\cdot \alpha] \blacktriangleright \sigma$.

(1.10) $\alpha_1, \ldots, \alpha_n$: **term** $(n \geqslant 1)$, β : **term**, $\alpha_1 \blacktriangleright \sigma$,
$[\alpha_2, \ldots, \alpha_n \cdot \beta] \blacktriangleright \tau \Rightarrow [\alpha_1, \ldots, \alpha_n \cdot \beta] \blacktriangleright \mathbf{s}\sigma\tau$.

(1.11) $\alpha_1, \ldots, \alpha_n$: **term** $(n \geqslant 0)$, $[\alpha_1, \ldots, \alpha_n \cdot 0] \blacktriangleright \sigma$
$\Rightarrow [\alpha_1, \ldots, \alpha_n] \blacktriangleright \sigma$.

(1.12) α : **unit** $\blacktriangleright \sigma$, β : **conslist** $\blacktriangleright \tau \Rightarrow \alpha\beta \blacktriangleright \mathbf{c}\sigma\tau$.

(1.13) α : **unit** $\blacktriangleright \sigma$, β : **snoclist** $\blacktriangleright \tau \Rightarrow \alpha\beta \blacktriangleright \mathbf{s}\sigma\tau$.

(1.14) The denotation of $\alpha\beta$ where α is a **function-symbol** and β is a **list** is defined each time a **function-symbol** is introduced. Here we introduce four **function-symbol**'s: *cons*, *snoc*, *car* and *cdr*. Let β : **list** $\blacktriangleright \tau$. Then

$cons\,\beta \blacktriangleright \mathbf{ca}\tau\mathbf{ad}\tau$,
$snoc\,\beta \blacktriangleright \mathbf{sa}\tau\mathbf{ad}\tau$,
$car\,\beta \blacktriangleright \mathbf{aa}\tau$,
$cdr\,\beta \blacktriangleright \mathbf{da}\tau$.

(2.1) **empty** $> \mathbf{0}$.

(2.2) α : **alphanumeric**, $\tilde{\alpha} \blacktriangleright \sigma$, β : **alphanumericseq** $> \tau \Rightarrow \alpha\beta > \mathbf{s}\sigma\tau$.

(3.1) $\rightarrow \mathbf{0} \blacktriangleright \mathbf{s00}$.

(3.2) $\rightarrow \mathbf{1} \blacktriangleright \mathbf{ss000}$.

(3.3) χ : **variable** $\blacktriangleright \sigma \Rightarrow \rightarrow\chi \blacktriangleright \mathbf{s}\sigma\mathbf{0}$.

(3.4) α : **literal** $\blacktriangleright \sigma \Rightarrow \rightarrow \alpha \blacktriangleright \mathbf{s}\sigma\mathbf{0}$.

(3.5) α : **metaliteral** $\blacktriangleright \sigma \Rightarrow \rightarrow\alpha \blacktriangleright \mathbf{s0}\sigma$.

(3.6) α : **term**, $\rightarrow \alpha \blacktriangleright \sigma \Rightarrow \rightarrow(\cdot \alpha) \blacktriangleright \sigma$ and $\rightarrow[\cdot \alpha] \blacktriangleright \sigma$.

(3.7) $\alpha_1, \ldots, \alpha_n$: **term** $(n \geqslant 1)$, β : **term**,
$\rightarrow\alpha_1 \blacktriangleright \mathbf{s}\sigma\tau$, $\rightarrow(\alpha_2, \ldots, \alpha_n \cdot \beta) \blacktriangleright \mathbf{s}\sigma'\tau'$
$\Rightarrow \rightarrow(\alpha_1, \ldots, \alpha_n \cdot \beta) \blacktriangleright \mathbf{sc}\sigma\sigma'\mathbf{c}\tau\tau'$

(3.8) $\alpha_1, \ldots, \alpha_n$: **term** $(n \geqslant 0)$, $\rightarrow(\alpha_1, \ldots, \alpha_n \cdot 0) \blacktriangleright \sigma$
$\Rightarrow \rightarrow(\alpha_1, \ldots, \alpha_n) \blacktriangleright \sigma$

(3.9) $\alpha_1, \ldots, \alpha_n$: **term** $(n \geqslant 1)$, β : **term**,
$\rightarrow\alpha_1 \blacktriangleright \mathbf{s}\sigma\tau$, $\rightarrow[\alpha_2, \ldots, \alpha_n \cdot \beta] \blacktriangleright \mathbf{s}\sigma'\tau'$
$\Rightarrow \rightarrow[\alpha_1, \ldots, \alpha_n \cdot \beta] \blacktriangleright \mathbf{ss}\sigma\sigma'\mathbf{c}\tau\tau'$

(3.10) $\alpha_1, \ldots, \alpha_n$: **term** $(n \geqslant 0)$, $\rightarrow[\alpha_1, \ldots, \alpha_n \cdot 0] \blacktriangleright \sigma$
$\Rightarrow \rightarrow[\alpha_1, \ldots, \alpha_n] \blacktriangleright \sigma$

(3.11) α : **unit**, $\rightarrow\alpha \blacktriangleright \mathbf{s}\sigma\tau$, β : **conslist**, $\rightarrow\beta \blacktriangleright \mathbf{s}\sigma'\tau'$
$\Rightarrow \rightarrow\alpha\beta \blacktriangleright \mathbf{sc}\sigma\sigma'\mathbf{c}\tau\tau'$

(3.12) α : **unit**, $\rightarrow\alpha \blacktriangleright \mathbf{s}\sigma\tau$, β : **snoclist**, $\rightarrow\beta \blacktriangleright \mathbf{s}\sigma'\tau'$
$\Rightarrow \rightarrow\alpha\beta \blacktriangleright \mathbf{ss}\sigma\sigma'\mathbf{c}\tau\tau'$

(3.13) $\alpha_1, \ldots, \alpha_n, \beta$: **const** $(n \geqslant 1)$,
$\rightarrow(\alpha_1, \ldots, \alpha_n \cdot \beta) \blacktriangleright \sigma \Rightarrow \alpha_1; \ldots; \alpha_n \rightarrow \beta \blacktriangleright \sigma$

Example 1.1.

(1) $x \blacktriangleright \sigma$, $y \blacktriangleright \tau \Rightarrow (x \cdot y) \blacktriangleright \mathbf{c}\sigma\tau$ and $[x \cdot y] \blacktriangleright \mathbf{s}\sigma$.

(2) $car(cons(1, 0)) \blacktriangleright \mathbf{s00}$.

The purpose of introducing terms is, of course, to *use* them as names of sexps. Thus, two terms are defined to be equal (=) if they denote the same sexp.

Note that, according to our definition, each term denotes a unique sexp (under a fixed environment). Note also that the denotation of a closed term (i.e., a term not containing variables) is independent of the environment.

Example 1.2.

(1) $car(cons(1, 0)) = 1$.

(2) $\mathrm{ab} = [[1, 1, 0, 0, 0, 0, 1], [1, 1, 0, 0, 0, 1, 0]]$.

(3) $cons(\mathrm{b}, \mathrm{c}) = (\mathrm{b} \cdot \mathrm{c})$.

(4) $\rightarrow \mathrm{car}[[\mathrm{X} \cdot \mathrm{Y}], \mathrm{X}] = [\mathrm{car}[[0 \cdot 0], 0] \cdot 0((\mathrm{x} \cdot \mathrm{y}), \mathrm{x})]$.

Note that a sequent always denote an atom σ. We call the car (cdr σ the *real part* (*meta part*, resp.) of the sequent. The meta part of a sequent extracts the occurrences of metaliterals in the sequent and the real part represents the remaining part of the sequent. Thus, metaliterals in a sequent will work as *schematic variables* in formal systems to be introduced in the next section.

2. Formal systems

In this section we introduce the concept of a formal system which plays a fundamental role in our study of the domain S. The basic idea of our definition of a formal system comes from Smullyan [11]. However, there is a crucial difference between our definition and Smullyan's. Namely, according to our definition, a formal system, which talks about sexps, is itself a sexp. This feature of a formal system enables us to establish metamathematical results such as Theorem 3.2 without the device of Gödel numbering.

Let A be a subset of S. We call A an *axiom system*. Relative to an axiom system A, we define *theorems* (of A) inductively as follows:

(1) (*axiom*) If $x \in A$ then x is a theorem,

(2) (*modus ponens*) If x and $(x \cdot y)$ are theorems then so is y.

Elements of A are called *axioms*. That x is a theorem of A is expressed by $\vdash x$ (in A) or simply $\vdash x$.

Example 2.1. Let A be the following set:

$$\{\mathrm{append}[0, y, y] \mid y \in \mathbf{S}\} \cup$$
$$\{(\mathrm{append}[x2, y, z2] \cdot \mathrm{append}[(x1 \cdot x2), y, (x1 \cdot z2)]) \mid x1, x2, y, z1, z2 \in \mathbf{S}\}.$$

Then, e.g., we have $\vdash \mathrm{append}[(\mathrm{a}, \mathrm{b}), (\mathrm{c}, \mathrm{d}), (\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d})]$.

According to the above definition, arbitrary subset of S becomes an axiom system. In order to define the concept of finitary axiom systems, we first define the concept of a formal system. We will call arbitrary sexp f a *formal system*. A formal system f determines an axiom system $Ax(f)$ in the following way. First, we define a binary relation $\in$ on sexps as follows:

(1) $x \in [x \cdot y]$,

(2) $z \in y \Rightarrow z \in [x \cdot y]$.

For instance, we have $z \in$ [a, b, c] iff z equals one of a, b or c. If f is a formal system and $z \in f$, we say that z is an *axiom schema* of f.

Example 2.2. If

$$f = [\rightarrow \text{append}[0, Y, Y],\ \text{append}[X2, Y, Z2] \rightarrow \text{append}[(X1 \cdot X2), Y, (X1 \cdot Z2)]]$$

then f has two axiom schemata:

$$\rightarrow \text{append}[0, Y, Y] = [\text{append}[0, 0, 0] \cdot 0(0, y, y)]$$

and

$$\text{append}[X2, Y, Z2] \rightarrow \text{append}[(X1 \cdot X2), Y, (X1 \cdot Z2)] = [(\text{append}[0, 0, 0] \cdot \text{append}[0, 0, 0]) \cdot (0(x2, y, z2), 0((x1 \cdot x2), y, (x1 \cdot z2)))].$$

Now, in order to obtain instances of an axiom schema, we introduce the concept of a formal environment. We will call any sexp a *formal environment*. We also introduce the function symbol *assoc*. The denotation of terms containing *assoc* is defined as follows:

(1) $assoc(x, 0) = 0$,

(2) $x = car(car(e)) \Rightarrow assoc(x, e) = cdr(car(e))$,

(3) $x \neq car(car(e)) \Rightarrow assoc(x, e) = assoc(x, cdr(e))$.

Note that according to the above definition, $assoc[0, 0]$, for instance, is undefined. For the sake of completness, we put $assoc\ \alpha = 0$ if $assoc\ \alpha$ is undefined under the above definition. Similar convention will be employed in the rest of the paper.

Example 2.3.

$$assoc(y, [(x \cdot a), (y \cdot b)]) = b,$$
$$assoc(z, [(x \cdot a), (y \cdot b)]) = 0$$

Next, we introduce the function symbol *instance*, such that $instance(z, e)$ gives the instantiation of z by e. We define *instance* by

$$instance(z, e) = subst(e, cdr(z), car(z))$$

where *subst* is defined by:

(1) $subst(e, 0, t) = t$,

(2) x:atom $\Rightarrow subst(e, x, t) = assoc(x\ e)$,

(3) $subst(e, (x1 \cdot x2), [t1 \cdot t2]) = [subst(e, x1, t1) \cdot subst(e, x2, t2)]$,

(4) $subst(e, (x1 \cdot x2), (t1 \cdot t2)) = (subst(e, x1, t1) \cdot subst(e, x2, t2))$.

Example 2.4.

$$instance(\text{append[X2, Y, Z2]} \rightarrow \text{append[(X1 · X2), Y, (X1 · Z2)]},$$
$$\text{[(x1 · a), (x2 · (b)), (y · (c, d)), (z2 · (b, c, d))])}$$
$$= (\text{append[(b), (c, d), (b, c, d)]} \cdot \text{append[(a, b), (c, d), (a, b, c, d)]}).$$

Now, for any formal system f we define Ax(f) as an axiom system whose axioms are all the instances of members of f. Namely, we put

$$\text{Ax}(f) = \{instance(z, e) \mid z \in f, e \in \mathbf{S}\}$$

Example 2.5. If f is the formal system of Example 2.2 then Ax(f) is the axiom system of Example 2.1.

An axiom system A is said to be *finitary* if $A = \text{Ax}(f)$ for some formal system f. We will write $\vdash z$ (in f) if $\vdash z$ (in Ax(f)).

Let f be a formal system,

$$c = a_1; \ldots : a_n \rightarrow b$$

be an axiom schema of f, and

$$z = (x_1, \ldots, x_n \cdot y)$$

be an instance of c. We will call each a_i a *premise* of c and b the *conclusion* of c. Similarly, each x_i will be called a premise of z and y the conclusion of z. It is easy to verify the following:

If each x_i $(1 \leq i \leq n)$ is a theorem of Ax(f), *then so is y.*

Let f, a be sexps. We define a subset $\langle a, f\rangle$ of **S** by

$$\langle a, f\rangle = \{x \mid \vdash a[x] \text{ (in } f)\}.$$

A subset X of **S** is said to be *recursively enumerable* (r.e.) if $X = \langle a, f\rangle$ for some f and a. A sexp a used in this way will be called a *predicate*. X is defined to be *recursive* if both X and $\mathbf{S} - X$ are r.e.

Example 2.6. Let $M = [\rightarrow \text{mole[(X · Y)]}]$ and $A = [\rightarrow \text{atom[[X · Y]]}]$. Then $\langle \text{mole}, M\rangle = \mathbf{M}$ and $\langle \text{atom}, A\rangle = \mathbf{A}$. Hence, **M** and **A** are recursive.

We list more examples of formal systems which will be frequently used in this paper.

Eq = [→eq[X, X]],

Ne = [→ne[(X · Y), [U · V]],
→ne[[X · Y], (U · V)],
ne[X, U]→ne[(X · Y), (U · V)],
ne[X, U]→ne[[X · Y], [U · V]],
ne[Y, V]→ne[(X · Y), (X · V)],
ne[Y, V]→ne[[X · Y], [X · V]]],

Atom = [→atom[[X · Y]]],

Mole = [→mole[(X · Y)]],

C*r = [→car[(X · Y), X],
→car[[X · Y], X],
→cdr[(X · Y), Y],
→cdr[[X · Y], Y],
car[X, Y]; car[Y, Z]→caar[X, Z],
cdr[X, Y]; car[Y, Z]→cadr[X, Z],
car[X, Y]; cadr[Y, Z]→cadar[X, Z],
cdr[X, Y]; cadr[Y, Z]→caddr[X, Z]],

Cons = [→cons[X, Y, (X · Y)]],

Snoc = [→snoc[X, Y, [X · Y]]].

3. The universal formal system

In this section, we will define a formal system **Univ**, which has the property:

$$\vdash \text{theorem}[z, f] \text{ (in } \mathbf{Univ}) \textit{ iff } \vdash z \text{ (in } f).$$

It also has the property that $\langle a, f\rangle, \mathbf{Univ}\rangle = \langle a, f\rangle$. Thus, any r.e. set is represented by some sexp in the system **Univ**. Hence **Univ** will be called the *universal formal system*. First we prepare some auxiliary notions. Let X be a subset of **S**. For any natural number n, we define the set $[X]^n$ inductively as follows:

(1) $[X]^0 = \{0\}$,

(2) $[X]^{n+1} = \{[x \cdot y] \mid x \in X, y \in [X]^n\}$.

We put $[X]^* = \bigcup_{n=0}^{\infty} [X]^n$. Then any sexp $x \in [X]^n$ can be uniquely written as $x = [x_1, \ldots, x_n]$. If we define the concatenation (&) of two sexps in $[X]^*$ by

$$[x_1, \ldots, x_m] \,\&\, [y_1, \ldots, y_n] = [x_1, \ldots, x_m, y_1, \ldots, y_n],$$

then $[X]^*$ becomes a free monoid generated by X. Let us consider how concatenation of several sexps in $[X]^*$ is formally represented. Let

Append =
[→append[0, 0],
append[X2, V2]; append2[X1, V2, V]→append[[X1 · X2], V],
→append2[0, Y, Y],
append2[X2, Y, V2]→append2[[X1 · X2], Y, [X1 · V2]],
append2[X2, Y, V2]→append2[(X1 · X2), Y, (X1 · V2)]].

Then we have

$x_1 \,\&\, x_2 \,\&\, \ldots \,\&\, x_n = z$ iff $\vdash$ append$[[x_1, x_2, \ldots, x_n], z]$
(in **Append**)

for any $x_1, \ldots, x_n \in [\mathbf{S}]^*$. This equivalence suggests us to extend the definition of & as follows. Namely, for any $x, y \in \mathbf{S}$ we put

$x \,\&\, y$ = the unique z such that $\vdash$ append$[[x, y], z]$ (in **Append**).

Now, **Univ** is defined as follows:

Univ = **Theorem** & **Axiom** & **Member** & **Instance** & **Subst** & **Assoc**
& **Eq** & **Ne** & **Atom** & **C*r**

where

Theorem = [axiom[A, F]→theorem[A, F],
theorem[A, F]; theorem[(A · B), F]→theorem[B, F],
theorem[A[X], F]→(A, F)[X],
theorem[A[X], F]→true[(A, F)[X]]],

Axiom = [member[A, F]; instance[B, A]→axiom[B, F]],

Member = [→member[X, [X · Y]],
member[Z, Y]→member[Z, [X · Y]]],

Instance = [car[A, T]; cdr[A, X]; subst[E, X, T, B]→instance[B, A]],

Subst = [→subst[E, 0, T, T],
atom[X]; assoc[X, E, V]→subst[E, X, T, V],
subst[E, X1, T1, V1]; subst[E, X2, T2, V2]→
subst[E, (X1 · X2), [T1 · T2], [V1 · V2]],
subst[E, X1, T1, V1]; subst[E, X2, T2, V2]→
subst[E, (X1 · X2), (T1 · T2), (V1 · V2)]]

Assoc = [→assoc[X, 0, 0],
caar[E, K]; eq[X, K]; cdar[E, V]→assoc[X, E, V],
caar[E, K]; ne[X, K]; cdr[E, R]; assoc[X, R, V]
→assoc[X, E, V]].

Note that **Atom**, **Eq**, **Ne** and **C*r** are already defined in Section 2.

We now have the following theorem which establishes the universality of **Univ**. In the following theorem, $\vdash$ means the provability in **Univ** unless specified to be otherwise.

Theorem 3.1.

(i) $\vdash \mathrm{atom}[x]$ *iff* $x \in \mathbf{A}$.
(ii) $\vdash \mathrm{car}[x, z]$ *iff* $car(x) = z$.
(iii) $\vdash \mathrm{cdr}[x, z]$ *iff* $cdr(x) = z$.
(iv) $\vdash \mathrm{caar}[x, z]$ *iff* $car(car(x)) = z$.
(v) $\vdash \mathrm{cdar}[x, z]$ *iff* $cdr(car(x)) = z$.
(vi) $\vdash \mathrm{eq}[x, y]$ *iff* $x = y$.
(vii) $\vdash \mathrm{ne}[x, y]$ *iff* $x \neq y$.
(viii) $\vdash \mathrm{member}[a, f]$ *iff* $a \in f$.
(ix) $\vdash \mathrm{assoc}[x, e, v]$ *iff* $assoc(x, e) = v$.
(x) $\vdash \mathrm{subst}[e, x, t, v]$ *iff* $subst(e, x, t) = v$.
(xi) $\vdash \mathrm{instance}[b, a]$ *iff* $b = instance(a, e)$ *for some* e.
(xii) $\vdash \mathrm{axiom}[a, f]$ *iff* $a \in \mathrm{Ax}(f)$.
(xiii) $\vdash \mathrm{theorem}[a, f]$ *iff* $\vdash a$ (in f).
(xiv) $\vdash (a, f)[x]$ *iff* $\vdash a[x]$ (in f).
(xv) $\vdash \mathrm{true}[(a, f)[x]]$ *iff* $\vdash a[x]$ (in f).

We can now prove the existence of a r.e. but nonrecursive subset of **S**. Let us put $T = \langle \mathrm{true}, \mathbf{Univ} \rangle$. T is r.e. by definition. Let X be a subset of **S**. A sexp x is called a *Gödel sexp* for X, if it has the property

$$x \in X \text{ iff } x \in T.$$

We will show that if X is r.e. then there exists a Gödel sexp for it. So, suppose $X = \langle a, A \rangle$. We define a formal system B by putting:

$$B = [a[\mathrm{X}[\mathrm{X}]] \rightarrow b[\mathrm{X}]] \,\&\, A$$

where $b = [A]$. (The choice of b may be arbitrary so long as it is not used in A. We will call such a b a *new* predicate.) Let $H = (b, B)$. Then for any sexp x, we have

$H[x] \in T$
iff $\vdash \mathrm{true}[H[x]]$ (in **Univ**)
iff $\vdash b[x]$ (in B)
iff $\vdash a[x[x]]$ (in A)
iff $x[x] \in X$.

So, letting $x = H$, we see that $H[H]$ is a Gödel sexp for X. We have thus seen that any r.e. set has a Gödel sexp. Now assume that $\mathbf{S} - T$ is r.e. Then it has a Gödel sexp g. I.e., $g \in T$ iff $g \in \mathbf{S} - T$. This is a contradiction. We have thus proved the following theorem.

Theorem 3.2. *The set* $T = \langle \text{true}, \textbf{Univ} \rangle$ *is recursively enumerable but not recursive.*

Remark. We remark that no Gödel numbering or coding is used in the proof of the above theorem. Compare our proof with the proof of the Theorem on page 15 of Smullyan [11].

4. The reference language

In this section we introduce yet another notation system for sexps, which is mainly used to express Hyperlisp programs. This notation system is called the *reference language*. We define the reference language in terms of a formal system **Ref** as follows. First of all, since an expression of the reference language is a sequence of syntactic marks, we represent it as a sexp in the set $[X]^*$ for a suitably chosen X. Namely, a syntactic mark is represented by a sexp in X and this representation is extended homomorphically to the expressions in the reference language. With this in mind, we now define the formal system **Ref**. In the following definition of **Ref**, we use the following notational convention.

The expression of the form:

$$\alpha_1; \ldots; \alpha_n; \text{append}[[\chi_1, \ldots, \chi_n], \chi]; \gamma_1; \ldots; \gamma_n \rightarrow \delta[\chi, \nu]$$

where

$n \geq 1$, $m \geq 0$ and
$\alpha_i = \beta_i[\chi_i, \mu_i]$ or $\alpha_i = \beta_i[\chi_i]$ and
$\chi_1, \ldots, \chi_n, \chi$ are distinct metaliterals

will be abbreviated as:

$$\pi_1; \ldots; \pi_n; \gamma_1; \ldots; \gamma_m \rightarrow \langle \delta \mid \nu \rangle$$

where

$\pi_i = \langle \beta_i \mid \mu_i \rangle$ if $\alpha_i = \beta_i[\chi_i, \mu_i]$ and
$\pi_i = \langle \beta_i \rangle$ if $\alpha_i = \beta_i[\chi_i]$.

The expression of the form:

$$\beta_1[\chi_1, \mu_1]; \ldots; \beta_n[\chi_n, \mu_n]; \text{append}[[\chi_1, \ldots, \chi_n], \chi]; \text{append}[[\mu_1, \ldots, \mu_n], \mu] \rightarrow \delta[\chi, \mu]$$

where

$\chi_1, \ldots, \chi_n, \chi, \mu_1, \ldots, \mu_n, \mu$ are distinct metaliterals

will be abbreviated as:

$$\langle \beta_1 \rangle \cdots \langle \beta_n \rangle \rightarrow \langle \delta \rangle.$$

Note that the metaliterals $\chi_1, \ldots, \chi_n, \chi$ etc. are lost in the abbreviation. Although different choices of metaliterals yield different formal systems, the axiom system determined by these formal systems is always the same. Hence, nothing essential is lost by this abbreviation.

Example 4.1.

(1) ⟨quote⟩⟨unit|[U · V]⟩→⟨unit|[1[U] · 0(V)]⟩

is an abbreviation of

quote[X1]; unit[X2, [U · V]]; append[[X1, X2], X]
→unit[X, [1[U] · 0(V)]].

(2) ⟨term⟩→⟨form⟩

is an abbreviation of

term[X1, T1]; append[[X1], X]; append[[T1], T]→form[X, T].

Here is the definition of **Ref**:

Ref = **Form** & **Term** & **Unit** & **List** & **Conslist** & **Snoclist**
& **Lambdaexp** & **Labelexp** & **Metaexp** & **Metaterm** & **Metalist**
& **Literal** & **Metaliteral** & **Alphanumeric** & **Specialchar**
& **Pureform** & **Denote** & **Assoc** & **Append** & **Atom** & **Eq** & **Ne**

where

Form = [⟨term⟩→⟨form⟩,
⟨term|[T1 · U1]⟩⟨colon⟩⟨term|[T2 · U2]⟩→
⟨form|[(T1, T2) · (U1, U2)]⟩],

Term = [⟨unit⟩→⟨term⟩,
⟨term|[F · G]⟩⟨conslist|[X · Y]⟩→
⟨term|[(F · X) · (G · Y)]⟩,
⟨term|[F · G]⟩⟨snoclist|[X · Y]⟩→
⟨term|[[F · X] · (G · Y)]⟩],

Unit = [⟨literal|L⟩→⟨unit|[L · 0]⟩,
⟨metaliteral|M⟩→⟨unit|[0 · M]⟩,
⟨list⟩→⟨unit⟩,
⟨quote⟩⟨unit|[U · V]⟩→⟨unit|[1[U] · 0(V)]⟩,
⟨lambdaexp⟩→⟨unit⟩,
⟨labelexp⟩→⟨unit⟩,
⟨zero⟩→⟨unit⟩,
⟨one⟩→⟨unit⟩],

Literal = [⟨lowercase⟩⟨tail⟩→⟨literal⟩],

Metaliteral =
[⟨uppercase | U⟩⟨tail | T⟩; lower[U, L]; append[[L, T], V] →
⟨metaliteral | V⟩,
→ lower[[[1, 0 · X]], [[1, 1 · X]]]],

Alphanumeric =
[⟨empty⟩ → ⟨tail⟩,
⟨alphanumeric⟩⟨tail⟩ → ⟨tail⟩,
⟨digit⟩ → ⟨alphanumeric⟩,
⟨lowercase⟩ → ⟨alphanumeric⟩,
⟨uppercase⟩ → ⟨alphanumeric⟩,
→ empty[0, 0],
between[[0, 1, 1, 0, 0, 0, 0], [0, 1, 1, 1, 0, 0, 1], X] →
digit[[X], [X]],
between[[1, 1, 0, 0, 0, 0, 1], [1, 1, 1, 1, 0, 1, 0], X] →
lowercase[[X], [X]],
between[[1, 0, 0, 0, 0, 0, 1], [1, 0, 1, 1, 0, 1, 0], X] →
uppercase[[X], [X]],
ascii[Y]; le[X, Y]; le[Y, Z] → between[X, Z, Y],
bit[B1]; bit[B2]; bit[B3]; bit[B4]; bit[B5]; bit[B6]; bit[B7] →
ascii[[B1, B2, B3, B4, B5, B6, B7]],
→ bit[0],
→ bit[1],
→ lt[0, 1],
lt[X1, Y1] → lt[[X1 · X2], [Y1 · Y2]],
eq[X1, Y1]; lt[X2, Y2] → lt[[X1 · X2], [Y1 · Y2]],
eq[X, Y] → le[X, Y],
lt[X, Y] → le[X, Y]],

List = [⟨conslist⟩ → ⟨list⟩,
⟨snoclist⟩ → ⟨list⟩],

Conslist =
[⟨dot⟩⟨form | E⟩ → ⟨consdotseq1 | E⟩,
⟨form | [X · Y]⟩⟨dot⟩⟨form | [E · F]⟩ →
⟨consdotseq2 | [(X · E) · (Y · F)]⟩,
⟨form | [X · Y]⟩⟨punc⟩⟨consdotseq2 | [E · F]⟩ →
⟨consdotseq2 | [(X · E) · (Y · F)]⟩,
⟨consdotseq1⟩ → ⟨consdotseq⟩,
⟨consdotseq2⟩ → ⟨consdotseq⟩,
⟨consopen⟩⟨condotseq | X⟩⟨consclose⟩ → ⟨conslist | X⟩,
dot[D]; zero[Z, [0 · 0]]; append[[C, D, Z], E]; consdotseq[E, X] →
consseq[C, X],
⟨consopen⟩⟨consseq | X⟩⟨consclose⟩ → ⟨conslist | X⟩],

Snoclist =
[⟨dot⟩⟨form|E⟩→⟨snocdotseq1|E⟩,
⟨form|[X · Y]⟩⟨dot⟩⟨form|E · F]⟩→
⟨snocdotseq2|[[X · E] · (Y · F)]⟩,
⟨form|[X · Y]⟩⟨punc⟩⟨snocdotseq2|[E · F]⟩→
⟨snocdotseq2|[[X · E] · (Y · F)]⟩,
⟨snocdotseq1⟩→⟨snocdotseq⟩,
⟨snocdotseq2⟩→⟨snocdotseq⟩,
⟨snocopen⟩⟨snocdotseq|X⟩⟨snocclose⟩→⟨snoclist|X⟩,
dot[D]; zero[Z, [0 · 0]]; append[[C, D, Z], E]; snocdotseq[E, X]→
snocseq[C, X],
⟨snocopen⟩⟨snocseq|X⟩⟨snocclose⟩→⟨snoclist|X⟩],

Lambdaexp = [⟨lambda|L⟩⟨metaexp|[R · M]⟩→
⟨lambdaexp|[(L · R) · (0 · M)]⟩],

Labelexp = [⟨label|L⟩⟨metaexp|[R · M]⟩→
⟨labelexp|[(L · R) · (0 · M)]⟩],

Metaexp =
[⟨consopen⟩⟨metaterm|X⟩⟨punc⟩⟨form|[R · M]⟩⟨consclose⟩;
abstract[X, M, [A · B]]→⟨metaexp|[(A, R) · (0, B)]⟩,
→abstract[X, 0, [0 · 0]],
atom[M]; assoc[M, X, (M · A)]→abstract[X, M, [[A] · 0]],
atom[M]; assoc[M, X, 0]→abstract[X, M, [0 · M]],
abstract[X, M1, [A1 · B1]]; abstract[X, M2, [A2 · B2]]→
abstract[X, (M1 · M2), [(A1 · A2) · (B1 · B2)]]],

Assoc = [→assoc[X, 0, 0],
→assoc[X, [(X · V) · R], (X · V)],
ne[X, K]; assoc[X, R, V]→assoc[X, [(K · U) · R], V]],

Metaterm =
[⟨metaunit⟩→⟨metaterm⟩,
⟨metaunit|U⟩⟨equal⟩⟨metaterm|T⟩; append[[U, T], V]→
⟨metaterm|V⟩],

Metaunit = [⟨metaliteral|L⟩→⟨metaunit|[(L · 1)]⟩,
⟨metalist⟩→⟨metaunit⟩,
zero[Z, [0 · 0]]→metaunit[Z, 0]],

Metalist =
[⟨dot⟩⟨metaterm|T⟩→⟨metadotseq1|T⟩,
⟨metaterm|T⟩⟨dot⟩⟨metaterm|U⟩; left[T, Tl]; right[U, Ur];
append[[Tl, Ur], V]→⟨metadotseq2|V⟩,
⟨metaterm|T⟩⟨punc⟩⟨metadotseq2|U⟩; left[T, Tl]; right[U, Ur];
append[[Tl, Ur], V]→⟨metadotseq2|V⟩,
→left[0, 0],

left[R, Rl] → left[[(K · V) · R], [(K · (V · 0)) Rl]],
→ right[0, 0],
right[R, Rr] → right[[(K · V) · R], [(K · (0 · V)) · Rr]],
⟨metadotseq1⟩ → ⟨metadotseq⟩,
⟨metadotseq2⟩ → ⟨metadotseq⟩,
⟨open⟩⟨metadotseq | X⟩⟨close⟩ → ⟨metalist | X⟩,
dot[D]; zero[Z, [0 · 0]]; append[[C, D, Z], E]; metadotseq[E, X] →
metaseq[C, X],
⟨open⟩⟨metaseq | X⟩⟨close⟩ → ⟨metalist | X⟩,
consopen[X] → open[X],
snocopen[X] → open[X],
consclose[X] → close[X],
snocclose[X] → close[X]],

Specialchar =
| → zero[[[0, 1, 1, 0, 0, 0, 0], [0 · 0]],
→ one[[[0, 1, 1, 0, 0, 0, 0], [1 · 0]],
→ consopen[[[0, 1, 0, 1, 0, 0, 0]]],
→ consclose[[[0, 1, 0, 1, 0, 0, 1]]],
→ snocopen[[[1, 0, 1, 1, 0, 1, 1]]],
→ snocclose[[[1, 0, 1, 1, 1, 0, 1]]],
→ dot[[[0, 1, 0, 1, 1, 1, 0]]],
→ punc[[[0, 1, 0, 1, 1, 0, 0]]],
→ lambda[[[1, 0, 1, 1, 1, 0, 0]],
[[1, 1, 0, 1, 1, 0, 0], [1, 1, 0, 0, 0, 0, 1], [1, 1, 0, 1, 1, 0, 1],
[1, 1, 0, 0, 0, 1, 0], [1, 1, 0, 0, 1, 0, 0], [1, 1, 0, 0, 0, 0, 1]]],
→ label[[[0, 1, 0, 1, 1, 1, 1],
[[1, 1, 0, 1, 1, 0, 0], [1, 1, 0, 0, 0, 0, 1], [1, 1, 0, 0, 0, 1, 0],
[1, 1, 0, 0, 1, 0, 1], [1, 1, 0, 1, 1, 0, 0]]],
→ colon[[[0, 1, 1, 1, 0, 1, 0]]],
→ equal[[[0, 1, 1, 1, 1, 0, 1]]],
→ quote[[[0, 1, 0, 0, 1, 1, 1]]]],

Pureform = [form[E, [R · 0]] → pureform[E]],

Denote =
[form[F, X]; instance[X, E, V] → denote[F, E, V],
→ instance[[R · 0], E, R],
atom[M]; assoc[M, E, (N · V)] → instance[[R · M], E, V],
instance[[R1 · M1], E, V1]; instance[[R2 · M2], E, V2] →
instance[[(R1 · R2) · (M1 · M2)], E, (V1 · V2)],
instance[[R1 · M1], E, V1]; instance[[R2 · M2], E, V2] →
instance[[[R1 · R2] · (M1 · M2)], E, [V1 · V2]]].

We note that **Append**, **Atom**, **Eq** and **Ne** are already defined.

We will explain some properties of **Ref**, by examples. At the same time, we define the reference language in terms of **Ref**. The alphabet $\mathbf{L}_{ref}$ of the reference language consists of the following 74 characters.

(1) **lowercase:**

a b c d e f g h i j k l m n o p q r s t u v w x y z

(2) **uppercase:**

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(3) **digit**

0 1 2 3 4 5 6 7 8 9

(4) **special character:**

' ( ) , ; . : = [ ] λ Λ

The characters in $\mathbf{L}_{ref}$ are represented by the sexps in the set

$$X = \{x \mid \vdash \text{ascii}[x] : \textbf{Alphanumeric}\}.$$

Let α be a character in the alphabet. Then $[\![\alpha]\!] = x$ will mean that the character α is represented by the sexp $x \in X$ (or, x is the *code* of α). The coding is defined as follows:

First, we map the 74 characters in the alphabet into the ASCII character set. This mapping is identity, except for the following:

(i) ; is mapped to ,

(ii) λ is mapped to \

(iii) Λ is mapped to /

These mapped characters are then coded into X using their 7 bit ASCII codes. Here are some examples:

$$[\![\mathrm{U}]\!] = [1, 0, 1, 0, 1, 0, 1],$$

$$[\![\,,]\!] = [\![\,;]\!] = [0, 1, 0, 1, 1, 0, 0],$$

$$[\![\lambda]\!] = [1, 0, 1, 1, 1, 0, 0],$$

$$[\![\Lambda]\!] = [0, 1, 0, 1, 1, 1, 1].$$

Let $\mathbf{L}^*_{ref}$ be the set of all the finite sequences of characters in $\mathbf{L}_{ref}$. Then the above coding naturally extends homomorphically to

$$[\ ] : \mathbf{L}^*_{ref} \to [X]^*$$

by putting

$$[\alpha_1 \cdots \alpha_n] = [x_1, \ldots, x_n]$$

where

$$\alpha_i \in \mathbf{L}_{ref} \quad \text{and} \quad [\![\alpha_i]\!] = x_i.$$

Example 4.2. We have

α: lowercase *iff* $\vdash$ lowercase [⌈α⌉, ⌈α⌉],

α: uppercase *iff* $\vdash$ uppercase [⌈α⌉, ⌈α⌉],

α: digit *iff* $\vdash$ digit[⌈α⌉, ⌈α⌉],

$\vdash$ consopen [⌈(⌉],

$\vdash$ lambda[⌈λ⌉, lambda], etc.

Remark. The notation consopen[⌈(⌉] etc., in the above example, are strictly speaking illegal, since the syntactic marks ⌈ and ⌉ are not contained in the alphabet $\mathbf{L}_{mini}$ of the mini language. However the meaning of these expressions should be clear. At this point, we also remark that, after defining the reference language, we will have two notation systems for sexps. Namely, the mini language and the reference language. Since there is considerable overlap between the alphabets of these two languages, it will be sometimes ambiguous whether a particular expression is written in the mini language or in the reference language. We will rely on the context to determine which language is actually used. Such ambiguous expressions will, in most cases, have the same denotations in any of the two languages.

Among the strings in $\mathbf{L}^*_{ref}$, only some are grammatically meaningful. Such grammatically meaningful expressions are determined by the *predicates* in **Ref** as follows. An expression α in $\mathbf{L}^*_{ref}$ is said to be a *form* if $\vdash$ form[⌈α⌉, x] for some x. Other *syntactic categories* such as term, unit, literal, list etc. are defined similarly. The meaning of the second argument of the predicates form etc. will be explained later.

We summarize the syntax of **Ref** in terms of BNF as follows.

form ::= **term** | **term** : **term**

term ::= **unit** | **term list**

unit ::= **literal** | **metaliteral** | **list** | ' **unit** |
lambdaexp | **labelexp** | 0 | 1

list ::= **conslist** | **snoclist**

conslist ::= (**form**, . . . , **form** · **form**) | (**form**, . . . , **form**)

snoclist ::= [**form**, . . . , **form** · **form**] | [**form**, . . . , **form**]

lambdaexp ::= λ (**metaterm**; **form**)

labelexp ::= .l(**metaterm**; **form**)

metaterm ::= **metaunit** | **metaunit** = **metaterm**

metaunit ::= **metaliteral** | **metalist** | 0

metalist ::= **open metaterm, . . . , metaterm · metaterm close** |
open metaterm, . . . , metaterm close

open ::= (|[

close ::=)|]

Let χ be a metaliteral and let α be a form. An occurrence of χ in α is said to be *bound* if it occurs in a part of α of the form $\lambda(\pi;\beta)$ or $\Lambda(\pi;\beta)$ where π contains an occurrence of χ. An occurrence of χ in α is *free* if it is not bound. A form is *closed* if it contains no free occurrences of metaliterals. It is *open* otherwise.

Example 4.3. Let

$\alpha \simeq$ cond[null[X]: 'Y; '1: snoc('X1, Append[X2, Y])],
$\beta \simeq \lambda$([X = [X1 · X2], Y]; α),
$\gamma \simeq \Lambda$(Append; β).

Then α, β and γ are forms. All the occurrences of X, X1, X2, Y and Append in α are free. All the occurrences of X, X1, X2 and Y in β are bound, but Append is free in β. The form γ is closed.

The concept of a closed form is formalized in **Ref** as follows.

A form α is closed iff $\vdash$ pureform[$\lceil\alpha\rceil$].

Let us now explain how the denotation of a form is defined in **Ref**. First, we have the following theorem in **Ref**.

Theorem 4.1. (i) $\vdash$ form[f, x], $\vdash$ form[f, y] $\Rightarrow x = y$.
(ii) $\vdash$ form[f, x], $\vdash$ instance[x, e, u], $\vdash$ instance[x, e, v] $\Rightarrow u = v$.

For any sexp r and $i \geq 1$, we define the ith component of r (notation, $r.i$) as follows. The first component of r is the car of r, and the $i+1$st component of r is the car of the ith component of r. Now, let α be a form and suppose $\vdash$ form[$\lceil\alpha\rceil$, [$r \cdot m$]]. We remark that $r(m)$ roughly corresponds to the real part (meta part, resp.) of a sequent. If α is closed then we have $m = 0$ by **Pureform**.1. If α is open then $m \neq 0$ and m represents the position of free occurrences of metaliterals in α. In this case, given a formal environment e, the denotation of α (under e) is determined by *instantiating* [$r \cdot m$] by e. Namely, we say that α denotes v (under e) if $\vdash$ instance[[$r \cdot m$], e, v].

To proceed further with the explanation, we extend the alphabet $\mathbf{L}_{cf}$ by adjoining to it the following set of variables:

$x, y, z, \ldots$

We assume a one-to-one correspondence between the variables and metaliterals, e.g.,

$x \leftrightarrow$ X, $y \leftrightarrow$ Y, $z \leftrightarrow$ Z etc.

Given a form α, we define its *closure* β as the expression which results from α by replacing each free occurrence of a metaliteral in α by the corresponding variable. For instance, if $\alpha \simeq \lambda([Y]; \text{cons}[X, Y])$ then $\beta \simeq \lambda([Y]; \text{cons}[x, Y])$. The closure of a form will also be called a *cform*. Let α be a form, β its closure and e a formal environment. From e we obtain an environment Δ as follows. Let χ be a variable and χ' its corresponding metaliteral. Then we put $\Delta\chi = v$ where v is the unique sexp such that $\vdash \text{assoc}[\chi', e, v]$.

Example 4.4. If $e = [(x \cdot a), (y \cdot b)]$ then $\Delta x = a$, $\Delta y = b$ and $\Delta\chi = 0$ for other variables χ.

We then say that β denotes v (under Δ) if α denotes v (under e). Two cforms are defined to be equal (=) if they denote the same sexp.

Example 4.5. We list some identities between cforms, which are easily verifiable by the definition of **Ref**.

(i) $(x, y, z) = (x, y, z \cdot 0) = (x \cdot (y \cdot (z \cdot 0)))$,
(ii) $x : y = (x, y)$,
(iii) $f[x, y] = [f, x, y]$,
(iv) $'x = 1[x]$,
(v) $'1 : f[x] = ('1, f[x])$,
(vi) $f[x][y] = [f, x][y] = [[f, x], y]$.

See **Conslist** for (i), **Form** · 2 for (ii), **Term** · 3 for (iii), **Unit** · 4 for (iv).

Since the denotation of a cform containing bound variables are defined from the semantical motivation, we will explain it in the next section.

5. Semantics of Hyperlisp

In this section we define the semantics of Hyperlisp in terms of a formal system **Hy**.

5.1. The formal system Hy

Recall that for each $n > 0$ we have defined a subset $[\mathbf{S}]^n$ of **S**. It is easy to verify that $[\mathbf{S}]^n$ is recursive for each n. A subset Γ of **S** is said to be an n-ary function on **S** (written $\Gamma : [\mathbf{S}]^n \to \mathbf{S}$) if

(i) $\Gamma \subset [\mathbf{S}]^{n+1}$ and

(ii) for each $[x_1, \ldots, x_n] \in [\mathbf{S}]^n$ there uniquely exists a sexp z such that $[x_1, \ldots, x_n, z] \in \Gamma$.

If instead of (ii) we have

(iii) for each $[x_1, \ldots, x_n] \in \Gamma$ there exists *at most* one z such that $[x_1, \ldots, x_n, z] \in \Gamma$,

then we say that Γ is an n-ary *partial* function on **S** (written $\Gamma : [\mathbf{S}]^n \to \mathbf{S}$). If Γ is an n-ary (partial) function, we will write $\Gamma[x_1, \ldots, x_n] = z$ in place of $[x_1, \ldots, x_n, z] \in \Gamma$.

Let f be a formal system, and a be a predicate. For each $n \geq 0$, we put

$$\langle a, f\rangle^n = \{[x_1, \ldots, x_n] \mid \vdash a[x_1, \ldots, x_n] \text{ (in } f)\}.$$

We can easily verify that

(1) For each $a, f \in \mathbf{S}$ and $n > 0$, we can find $b, g \in \mathbf{S}$ such that $\langle a, f\rangle^n = \langle b, g\rangle$.

(2) For each $b, g \in \mathbf{S}$ and $n > 0$, we can find $a, f \in \mathbf{S}$ such that $\langle a, f\rangle^n = \langle b, g\rangle \cap [\mathbf{S}]^n$.

In fact, to prove (1), we just put

$$g = [a[\chi_1, \ldots, \chi_n] \to b[[\chi_1, \ldots, \chi_n]]] \,\&\, f$$

where b is a new predicate (e.g., $b = [f]$), and $\chi_1, \ldots, \chi_n$ are distinct metaliterals. (2) is proved by putting

$$f = [b[[\chi_1, \ldots, \chi_n]] \to a[\chi_1, \ldots, \chi_n]] \,\&\, g$$

where a is a new predicate, and $\chi_1, \ldots, \chi_n$ are distinct metaliterals. Thus, we see that $\Gamma \subset [\mathbf{S}]^n$ is r.e. iff $\Gamma = \langle a, f\rangle^n$ for some $a, f \in \mathbf{S}$. We can also prove that $\Gamma : [\mathbf{S}]^n \to \mathbf{S}$ is *recursive* if it is r.e. (We omit the proof since it is easy and quite similar to the proof of Theorem 4 in Chapter II of Smullyan [11].) A partial function $\Gamma : [\mathbf{S}]^n \to \mathbf{S}$ is said to be *partial recursive* if it is r.e.

The semantics of Hyperlisp will be given by a unary partial function $eval : [\mathbf{S}]^1 \to \mathbf{S}$. The partial function *eval* is defined by the formal system **Hy** given below as

$$eval = \langle \text{eval}, \mathbf{Hy}\rangle^2.$$

Since **Hy** has the property

$$\text{if } \vdash \text{eval}[e, u] \text{ and } \vdash \text{eval}[e, v] \text{ then } u = v,$$

eval, in fact, becomes a unary partial function. We will say that a sexp e *evaluates to* z if we have $\vdash \text{eval}[e, z]$.

Here is the definition of **Hy**:

Hy = **Eval** & **Evlis** & **Apply** & **Evcon** & **Subst** & **Point**
& **Eq** & **Ne** & **Atom** & **Mole** & **C*r**

where

Eval = [apply[F, X, V] → eval[[F · X], V],
evlis[X, Y]; apply[F, Y, V] → eval[(F · X), V]].

Evlis = [→evlis[0, 0],
ne[X, 0]; car[X, X1]; eval[X1, V1]; cdr[X, X2]; evlis[X2, V2] →
evlis[X, (V1 · V2)]],

Apply =
[→apply[0, X, 0],
car[X, X1]→apply[1, X, X1];
car[X, X1]; cadr[X, X2]; eq[X1, X2]→apply[eq, X, 1],
car[X, X1]; cadr[X, X2]; ne[X1, X2]→apply[eq, X, 0],
evcon[X, V]→apply[cond, X, V],
atom[F]; ne[F, 1]; ne[F, eq]; ne[F, cond]; eval[F, G]; apply[G, X, V]→
apply[F, X, V],
eq[F, (lambda · M)]; car[M, P]; cadr[M, B]; subst[X, P, B, E]; eval[E, V]
→apply[F, X, V],
eq[F, (label · M)]; car[M, P]; cadr[M, B]; subst[F, P, B, G]; apply[G, X, V]
→apply[F, X, V],
mole[F]; ne[F, 0]; car[F, L]; ne[L, lambda]; ne[L, label]; eval[F, G];
apply[G, X, V]→apply[F, X, V]],

Evcon =
[evcon[0, 0],
ne[X, 0]; caar[X, C]; eval[C, V1]; atom[V1]; cadar[X, E]; eval[E, V]→
evcon[X, V],
ne[X, 0]; caar[X, C]; eval[C, V1]; mole[V1]; cdr[X, Y]; evcon[Y, V]→
evcon[X, V]],

Subst =
[→subst[X, 0, B, B],
point[X, W, V]→subst[X, [W · U], B, V],
ne[(P1 · P2), 0]; subst[X, P1, B1, U1]; subst[X, P2, B2, V2]→
subst[X, (P1 · P2), (B1 · B2), (V1 · V2)],
ne(P1 · P2), 0]; subst[X, P1, B1, V1]; subst[X, P2, B2, V2]→
subst[X, (P1 · P2), [B1 · B2], [V1 · V2]]],

Point =
[→point[X, 0, 0],
atom[W]→point[X, W, X],
ne[W1, 0]; car[X, X1]; point[X1, W1, V]→point[X, (W1 · 0), V],
ne[W2, 0]; cdr[X, X2]; point[X2, W2, V]→point[X, (W1 · W2), V]].

We now introduce the concept of a proof tree. Consider the following formal system **Tree**:

Tree = [treeseq[S]→tree[[R · S]],
→treeseq[0],
tree[T]; treeseq[S]→treeseq[[T · S]],
→root[R, [R · S]],
→subtree[T, T],
member[T, S]→subtree[T, [R · S]],

member[T, S]; subtree[T1, T] → subtree[T1, [R · S]],
subtree[T1, T]; root[R, T1] → node[R, T],
→ leaf[L, [L]],
subtree[T1, T]; leaf[L, T1] → leaf[L, T],
member[X, [X · Y]],
member[Z, Y] → member[Z, [X · Y]]].

We will say that:

t is a *tree* if $\vdash$ tree[t],

r is the *root* of t if $\vdash$ root[r, t],

s is a *subtree* of t if $\vdash$ subtree[s, t],

n is a *node* of t if $\vdash$ node[n, t],

l is a leaf of t if $\vdash$ leaf[l, t].

Let X be a subset of **S**. A tree t is an *X-tree* if all of its nodes are in X. Remark that a general form of a tree t is

$$t = [r; t_1, \ldots, t_n] \quad (n \geq 0)$$

where r is the root of t and t_i are immediate subtrees of t.

A sexp x is said to be *elementary* if it is of the form

$$x = (x_1, \ldots, x_n \cdot x_{n+1}) \quad (n \geq 0)$$

where each x_i is an atom. An axiom system is *elementary* if each of its members is *elementary*. A formal system f is defined to be *elementary* if Ax(f) is elementary. Note that all the formal systems we have introduced so far are elementary. We will continue to consider only elementary formal systems.

Let f be an elementary formal system. We define the concept

P is a *proof tree* of z in f, where z is an atom

(written $P \vdash z$ (in f)) inductively as follows:

$$x = (x_1, \ldots, x_n \cdot x_{n+1}) \in \mathrm{Ax}(f)(n > 0),\ P_i \vdash x_i\ (\text{in } f)\ (1 \leq i \leq n)$$
$$\Rightarrow [x_{n+1}; P_1, \ldots, P_n] \vdash x_{n+1}\ (\text{in } f).$$

We note that if $P \vdash z$ (in f) then P is an **A**-tree and $car(P) = z$. Note also that $[z] \vdash z$ (in f) if $z \in \mathrm{Ax}(f) \cap \mathbf{A}$. We will omit (in f) when f is clear from the context. We also remark that, by modifying the universal system **Univ** slightly, one can easily obtain a formal system V such that

$$\vdash \mathrm{proof}[p, z, f]\ (\text{in } V) \text{ iff } p \vdash z\ (\text{in } f)$$

where f is an elementary formal system. It is also easy to verify that:

If f is an elementary formal system then
$\vdash z$ (in f) *iff* $P \vdash z$ (in f) *for some* P.

Let us now focus our attention to the elementary formal system **Hy**. We have the following theorem in **Hy**.

Theorem 5.1. (i) $P \vdash z, Q \vdash z \Rightarrow P = Q$.
(ii) $P \vdash \text{eval}[e, u], Q \vdash \text{eval}[e, v] \Rightarrow P = Q$ *and* $u = v$.
(iii) $P \vdash \text{apply}[f, x, u], Q \vdash \text{apply}[f, x, v] \Rightarrow P = Q$ *and* $u = v$.

By (ii) of this theorem, eval determines a unary partial function. We summarize basic properties of **Hy** in the following theorem. We will write $e \rhd v$ if $\vdash \text{eval}[e, v]$.

Theorem 5.2. (i) $\vdash \text{apply}[f, x, v]$ *iff* $f[\cdot x] \rhd v$.
(ii) $\vdash \text{evlis}[x, y], \vdash \text{apply}[f, y, v]$ *iff* $f(\cdot x) \rhd v$.
(iii) $x \rhd y_i (1 \leq i \leq n, n \geq 0)$ *iff* $\vdash \text{evlis}[(x_1, \ldots, x_n), (y_1, \ldots, y_n)]$.
(iv) $0 \rhd 0$.
(v) $'x \rhd x$.
(vi) $x = y$ *iff* $\text{eq}[x, y] \rhd 1$.
(vii) $x \neq y$ *iff* $\text{eq}[x, y] \rhd 0$.
(viii) $x_i \rhd z_i \in \mathbf{M} (1 \leq i < k \leq n, n \geq 0), x_k \rhd z_k \in \mathbf{A}, y_k \rhd z$
$\Rightarrow \text{cond}[x_1 : y_1; \ldots; x_n : y_n] \rhd z$.
(ix) $x_i \rhd z_i \in \mathbf{M} (1 \leq i \leq n, n \geq 0)$
$\Rightarrow \text{cond}[x_1 : y_1; \ldots; x_n : y_n] \rhd 0$.
(x) $\vdash \text{subst}[x, p, b, e], e \rhd v \Rightarrow \vdash \text{apply}[\text{lambda}(p, b), x, v]$.
(xi) $\vdash \text{subst}[\text{label}(p, b), p, b, g], \vdash \text{apply}[g, x, v]$
iff $\vdash \text{apply}[\text{label}(p, b), x, v]$.
(xii) $f \in \mathbf{A}, f \neq 1, f \neq \text{eq}, f \neq \text{cons}, f \rhd g, \vdash \text{apply}[g, x, v]$
$\Rightarrow \vdash \text{apply}[f, x, v]$.
(xiii) $f \in \boldsymbol{M}, f \neq 0, car(f) \neq \text{lambda}, car(f) \neq \text{label}, f \rhd g, \vdash \text{apply}[g, x, v]$
$\Rightarrow \vdash \text{apply}[f, x, v]$.

Proof. We prove only (iv) and (vi). Instead of giving proofs, we will explain the intended meaning.

(i), (ii): (i) and (ii) characterizes *eval* (in terms of *apply* and *evlis*). Suppose a form e is given to *eval* (as its first argument). The car of e is a *function* and (i) if e is an atom then the function is *applied* to the cdr of e, and (ii) if e is a molecule then the function is *applied* to the result of evaluating the argument by *evlis*.

(iii): *evlis* evaluates each component of the argument successively.

(iv): Since $0 = (0 \cdot 0)$, by (ii) and **Apply** $\cdot$ 1, 0 evaluates to 0.

(v): $'x = 1[x]$ and it *quotes* x. See **Apply** $\cdot$ 2.

(viii), (ix): A *conditional expression* usually takes the form

$$\text{cond}[x_1: y_1; \ldots; x_n: y_n].$$

The *conditions* x_i are evaluated successively until an atom z_k (an atom represents *true*) is returned. The corresponding form y_k is then evaluated and its results is returned as the value of the conditional expression. If all the conditions turn out to be *false* (a molecule represents false) then the value of the conditional expression is 0.

(x)–(xiii): These are explained later.

Proof of (*vi*): Since $x = y$, we have

```
[eval[eq[x, y], 1];
  [apply[eq, [x, y], 1];
    [car[[x, y], x]],
    [cadr[[x, y], y];
      [cdr[[x, y], [y]]],
      [car[[y], y]]
    ],
    [eq[x, y]]
  ]
]⊢ eval[eq[x, y], 1]  □
```

In Hyperlisp, function abstraction is realized by lambda expressions, and recursion is realized by label expressions. We explain lambda expressions first.

5.1.1. Lambda expressions

Consider a binary function II on S such that $II[x, y] = (y \cdot x)$. We remark that II is represented by $\langle \text{xcons}, X\rangle$[3], where

$$X = [\rightarrow \text{xcons}[\text{X}, \text{Y}, (\text{Y} \cdot \text{X})]].$$

We wish to find a sexp *xcons* such that

$$xcons[x, y] \triangleright (y \cdot x)$$

holds for all sexps x and y. Let us note that

$$1[(y \cdot x)] \triangleright (y \cdot x).$$

Therefore, if Bourbaki's notation (See Bourbaki [2] or Quine [8]) were available, we could put:

$$xcons = \lambda([\square, \square]; 1[(\square \cdot \square)]).$$

Then the evaluation of $xcons[x, y]$ could be as follows. The *argument* $[x, y]$ is first matched with the *parameter part* $[\square, \square]$ of *xcons*, and we have

$$([x, y]; 1[(\square \cdot \square)]).$$

Then x and y are sent to the *body* following the arrows, and we have: $1[(y \cdot x)]$. This sexp is then evaluated to $(y \cdot x)$.

In the actual Hyperlisp reference language, we use metaliterals instead of boxes and arrows. Thus, we put

$$xcons = \lambda([X, Y]; 1[Y \cdot X]).$$

We now explain the notation. First note that $([X, Y]; 1[(Y \cdot X)])$ is a metaexp since $[X, Y]$ is a metaterm and $1[(Y \cdot X)]$ is a form. (See the definition of **Ref** in Section 4.) The metaliteral $[X, Y]$ means that the occurrences of metaliterals X and Y in the metaexp are *bound*, and that X should be bound to the *car* of the argument and Y to the *cadr*. Formally, in **Ref** we have

(1) $\vdash$ metaterm[$\lceil$[X, Y]], [(x · (1)), (y · (0, 1))]].

Note that car of (1) is 1 and cadr of (0, 1) is also 1. As for the body, we have

(2) $\vdash$ form[$\lceil 1(Y \cdot X)$]], [1[(0 · 0)] · 0((y · x))]].

The *abstraction* of the metapart (i.e., cdr) of the body with respect to the metaterm is $[0(([(0, 1)] \cdot [(1)])) \cdot 0]$ since we have

(3) $\vdash$ abstract[[(x · (1)), (y · (0, 1))], 0((y · x)), [0(([(0, 1)] · [(1)])) · 0]].

The cdr of the abstraction is 0, and it means that the abstraction contains no free metaliterals. This is because all the free metaliterals in $1[(Y \cdot X)]$ become bound by the metaterm $[X, Y]$. The car of the abstraction is $0(([(0, 1)] \cdot [(1)]))$ and it contains two *boxes* (i.e., $[(0, 1)]$ and $[(1)]$) in the appropriate places. (Compare $0(([0, 1)] \cdot [(1)]))$ with $1[(\square \cdot \square)]$ in Bourbaki's notation.) The sexps contained in these boxes, i.e., (0, 1) and (1), say that cadr of the argument are to be substituted for the first box and car to the second.

By (1), (2) and (3) we have

(4) $\vdash$ metaexp[$\lceil([X, Y]; 1[Y \cdot X])]$], [$(p, b) \cdot 0$]]

where $p = 0(([0, 1)] \cdot [(1)]))$ and $b = 1[(0 \cdot 0)]$. (See the definition of **Metaexp**.) By (4) we have

(5) $\vdash$ lambdaexp[$\lceil \lambda([X, Y]; 1[(Y \cdot X)])]$, [lambda($p, b$) · 0]].

(See the definition of **Lambdaexp**.) Thus, we have

$$\lambda([X, Y]; 1[(Y \cdot X)]) = \text{lambda}(0(([(0, 1)] \cdot [(1)])), 1[(0 \cdot 0)]).$$

From the above construction, it is clear that the denotation of a lambda expression is independent of the choice of bound variables (i.e., metaliterals). For instance, we have

$$\lambda([X, Y]; 1[(Y \cdot X)]) = \lambda([U, V]; 1[(V \cdot U)]).$$

The evaluation of $xcons[x, y]$ goes like this:

$$
\begin{array}{l}
[\mathrm{eval}[xcons[x, y], (y \cdot x)]; \\
\quad [\mathrm{apply}[xcons, [x, y], (y \cdot x)]; \\
\quad\quad [\mathrm{eq}[xcons, \mathrm{lambda}(p, b)]], \\
\quad\quad [\mathrm{car}[(p, b), p]], \\
\quad\quad [\mathrm{cadr}[(p, b), b]; \ldots], \\
\quad\quad [\mathrm{subst}[[x, y], p, b, 1[y \cdot x)]]; \ldots], \\
\quad\quad [\mathrm{eval}[1[(y \cdot x)], (y \cdot x)]; \ldots] \\
\quad] \\
] \vdash \mathrm{eval}[xcons[x, y], (y \cdot x)].
\end{array}
$$

Thus, we have: $xcons[x, y] \triangleright (y \cdot x)$. The reader is urged to examine the definition of subst in **Hy**, and convince himself that subst selects and substitutes correct values into the body of the lambda expression obtaining necessary information from the parameter part of the lambda expression. We note that in Hyperlisp, unlike Lisp ([5, 6]), arguments are actually substituted for the variables in the application of lambda expressions.

We list some more examples.

Example 5.1.

$$
\begin{array}{l}
\lambda([\mathrm{X}]; '\lambda([\mathrm{Y}]; '(\mathrm{X} \cdot \mathrm{Y}))) = \\
\quad (\mathrm{lambda}, (0, (0, 0, (0, ([(1)])))), [1, (\mathrm{lambda}, (0, (0 \cdot [(1)]), [1, 0])]).
\end{array}
$$

This awful equation teaches us that the left-hand side of the equation is for human beings and the right-hand side is for computers. In fact, we have computed the right-hand side by a computer.

Example 5.2. (A *fixpoint* of eval)

$$[\lambda([\mathrm{X}]; '[\mathrm{X}, \mathrm{X}]), \lambda([\mathrm{X}]; '[\mathrm{X}, \mathrm{X}])] \triangleright [\lambda([\mathrm{X}]; '[\mathrm{X}, \mathrm{X}]), \lambda([\mathrm{X}]; '[\mathrm{X}, \mathrm{X}])]$$

Example 5.3. Let

$$
\begin{array}{l}
atom = \lambda([\mathrm{X} = [\mathrm{X1} \cdot \mathrm{X2}]]; \mathrm{eq}[\mathrm{X}, [\mathrm{X1} \cdot \mathrm{X2}]]), \\
mole = \lambda([\mathrm{X} = (\mathrm{X1} \cdot \mathrm{X2})]; \mathrm{eq}[\mathrm{X}, (\mathrm{X1} \cdot \mathrm{X2})]), \\
car = \lambda([[\mathrm{X} \cdot \mathrm{Y}]]; '\mathrm{X}), \\
cdr = \lambda([[\mathrm{X} \cdot \mathrm{Y}]]; '\mathrm{Y}), \\
cons = \lambda([\mathrm{X}, \mathrm{Y}]; '(\mathrm{X} \cdot \mathrm{Y})), \\
snoc = \lambda([\mathrm{X}, \mathrm{Y}]; '[\mathrm{X} \cdot \mathrm{Y}]).
\end{array}
$$

Then

$$atom[x] \rhd 1 \text{ iff } \vdash \text{atom}[x] \text{ (in } \textbf{Atom}\text{) iff } x \text{ is an atom,}$$

$$atom[x] \rhd 0 \text{ iff } \vdash \text{mole}[x] \text{ (in } \textbf{Mole}\text{) iff } x \text{ is a molecule,}$$

$$mole[x] \rhd 1 \text{ iff } \vdash \text{mole}[x] \text{ (in } \textbf{Mole}\text{) iff } x \text{ is a molecule,}$$

$$mole[x] \rhd 0 \text{ iff } \vdash \text{atom}[x] \text{ (in } \textbf{Atom}\text{) iff } x \text{ is an atom,}$$

$$car[x] \rhd z \text{ iff } \vdash \text{car}[x, z] \text{ (in } \textbf{C*r}\text{) iff } z = \mathbf{a}x,$$

$$cdr[x] \rhd z \text{ iff } \vdash \text{cdr}[x, z] \text{ (in } \textbf{C*r}\text{) iff } z = \mathbf{d}x,$$

$$cons[x, y] \rhd z \text{ iff } \vdash \text{cons}[x, y, z] \text{ (in } \textbf{Cons}\text{) iff } z = \mathbf{c}xy,$$

$$snoc[x, y] \rhd z \text{ iff } \vdash \text{snoc}[x, y, z] \text{ (in } \textbf{Snoc}\text{) iff } z = \mathbf{s}xy.$$

The metaterm [[X · Y]] in the definition of *car* and *cdr* above means that X represents the caar of the argument and Y the cdar. A metaterm may contain equality symbols as in: [X = [X1 · X2], Y]. In this case, both X and [X1 · X2] represents the car of the argument. See the definition of **Metaterm** for more details.

5.1.2. Label expressions

A label expression, usually, takes the form

$$\Lambda(\chi; \varepsilon)$$

where χ is a metaliteral and ε is a lambda expression. Let $\alpha \simeq \Lambda(\chi; \varepsilon)$ be a *closed* label expression, and let δ be the result of textually substituting α for all free occurrences of χ in ε. We abbreviate this as $\delta \simeq \varepsilon\{\alpha/\chi\}$. Suppose that α denotes a and δ denotes d. Then by **Apply** · 8 we have for any x and z,

$$\vdash \text{apply}[a, x, z] \text{ iff } \vdash \text{apply}[d, x, z].$$

We may thus say that a and d are *functionally equivalent* (written, $a \sim d$). For instance, if we put

$$\begin{aligned} append = \Lambda(&\text{Append}; \lambda([\text{X} = [\text{X1} \cdot \text{X2}], \text{Y}]; \\ &\text{cond}[\text{eq}[\text{X}, 0]: \text{'Y}; \\ &\quad \text{'1}: cons(\text{'X1}, \text{Append}[\text{X2}, \text{Y}])])) \end{aligned}$$

then we have

$$\begin{aligned} append \sim \lambda([\text{X} = [\text{X1} \cdot \text{X2}], \text{Y}]; \\ \text{cond}[\text{eq}[\text{X}, 0]: \text{'Y}; \\ \text{'1}: cons(\text{'X1}, append[\text{X2}, \text{Y}])]). \end{aligned}$$

It then follows that

(i) $append[0, y] \rhd y$,

(ii) $append[x2, y] \rhd z2 \Rightarrow append[(x1 \cdot x2), y] \rhd [x1 \cdot z2]$.

Thus we see that *append* represents append of the formal system f in Example 2.2. I.e.,

$$append[x, y] \rhd z \text{ iff } \vdash \text{append2}[x, y, z] \text{ (in } \textbf{Append}).$$

We remark that both in lambda and label expressions, arguments are actually substituted into the bodies so that we do not have *funarg problems* in Hyperlisp. (See, Allen [1] and McCarthy [7] for funarg problems.) Also since variables and other entities are clearly separated in Hyperlisp, quotation (') in Hyperlisp realizes Quine's quasi-quotation (Quine[8]). Because of quasi-quotation and our mechanism of lambda binding, most primitive functions are definable in Hyperlisp. (See Example 5.3.)

5.1.3. Computed functions

Consider the application of f to x. If f is an atom other than 1, eq or cond, or if f is a nonzero molecule whose *car* is not equal to lambda or label, then f is *evaluated* and the result (if exists) is applied to x. (See Theorem 5.2 (xii)–(xiii).) Such functions are called *computed functions*. For instance, we have

$$\lambda([X]; '\lambda([Y]; '(X \cdot Y)))[a][b] \rhd (a \cdot b).$$

We can simulate Curry's *paradoxical combinator* (Curry [3])

$$Y = \lambda h \cdot (\lambda x \cdot h(xx))(\lambda x \cdot h(xx))$$

using computed functions.

Let $y = \lambda([H]; \lambda([X]; H[X[X]])[\lambda([X]; H[X[X]])])$. Then

$$y[\lambda([\text{Append}]; '\lambda([X = [X1 \cdot X2], Y];$$
$$\text{cond}[\text{eq}[X\ 0]: 'Y;$$
$$'1: cons('X1, \text{Append}[X2, Y])])]$$

is functionally equivalent with *append* in 5.1.2. In this way, we can realize recursion without using labels.

5.2. Formal representability of functions

For any $g \in \mathbf{S}$ and $n \geq 0$, we put

$$\{g\}^n = \{[x_1, \ldots, x_n, z]; \vdash \text{apply}[g, [x_1, \ldots, x_n], z]\}.$$

By Theorem 5.1, we see that $\{g\}^n : [\mathbf{S}]^n \to \mathbf{S}$. If Γ is an n-ary partial function and $\Gamma = \{g\}^n$, we say g represents Γ. We are interested in the question

Is the meaning function eval representable?

The answer is affirmative by (ii) of the follov ng general theorem which says that the set of representable functions coincides with that of partial recursive functions.

Theorem 5.3. (i) *For any* $g \in \mathbf{S}$ *and* $n \geq 0$, *we can find* $a, f \in \mathbf{S}$ *such that*

$$\{g\}^n = \langle a, f\rangle^{n+1}.$$

(ii) *For any* $a, f \in \mathbf{S}$ *and* $n \geq 0$ *such that* $\langle a, f\rangle^{n+1} : [\mathbf{S}]^n \to \mathbf{S}$, *we can find* $a, g \in \mathbf{S}$ *such that*

$$\{g\}^n = \langle a, f\rangle^{n+1}.$$

A proof of this theorem will be given in part II of this paper. The sexp g given by this theorem such that $\{g\}^1 = eval$ is, however, far from being practical. We will give a simple and practical solution to the above question. Let $g = \lambda([X]; X)$. Then we have

$$\begin{aligned}&\{g\}^1[x] = z\\ &\text{iff} \vdash \text{apply}[g, [x], z]\\ &\text{iff} \vdash \text{eval}[x, z]\\ &\text{iff} \vdash eval[x] = z.\end{aligned}$$

We will call a sexp *universal* if it represents *eval*. We now give yet another example of a universal function which faithfully reflects the structure of **Hy**. In the following we use strings of bold faced letters as syntactic constants for forms. First we define several *open* forms:

```
*evlis = .1(Evlis; λ([X = [X1 · X2]];
            cond[null[X]: 0;
                 '1: cons(Eval[X1], Evlis[X2])])),

*evcon = .1(Evcon; λ([X = [[E1, E2] · X2];
            cond[null[X]: 0;
                 Eval[E1]: Eval[E2];
                 '1: Evcon[X2]])),

*apply = .1(Apply; λ([F = [L, P, B], X = [X1, X2]];
            cond[null[F]: 0;
                 atom[F]:
                   cond[eq[F, 1]: 'X1;
                        eq[F, eq]: eq[X1, X2];
                        eq[F, cond]: *evcon[X],
                        '1: Apply(Eval[F], 'X)];
                 eq[L, lambda]: Eval(subst[X, P, B]);
                 eq[L, label]: Apply(subst[F, P, B], 'X);
                 '1: Apply(Eval[F], 'X)]))
```

where

subst ≃
  Λ(Subst; λ([X, P = [P1 · P2], B = [B1 · B2]];
    cond[**null**[P]: 'B;
      atom[P]: **point**[X, P1];
      atom[B]: **snoc**(Subst[X, P1, B1], Subst[X, P2, B2]);
      '1: **cons**(Subst[X, P1, B1], Subst[X, P2, B2])])),

point ≃ Λ(Point; λ([X = [X1 · X2], W = [W1 · W2]];
    cond[**null**[W]: 0;
      atom[W]: 'X;
      null[W2]: Point[X1, W1];
      '1: Point[X2, W2]])),

atom ≃ λ([X = [X1 · X2]]; eq[X, [X1 · X2]]),

null ≃ λ([X]; eq[X, 0]),

cons ≃ λ([X, Y]; '(X · Y)),

snoc ≃ λ([X, Y]; '[X · Y]).

We then define the following *closed* forms:

eval ≃ Λ(Eval; λ([E = [F · X]];
    cond[**atom**[E]: ***apply**[F, X];
      '1: ***apply**('F, ***evlis**[X])])),

evlis ≃ ***evlis**{**eval**/Eval},

evcon ≃ ***evcon**{**eval**/Eval},

apply ≃ ***apply**{**eval**/Eval}

where **evlis**, e.g., is the result of textually substituting **eval** for all free occurrences of Eval in ***evlis** (recall Section 5.1.2). Then we have the following theorem which establishes the universality of eval.

Theorem 5.4. (i) ⊢ apply[**eval**, {e}, v] *iff* ⊢ eval[e, v],
(ii) ⊢ apply[**apply**, {f, x}, v] *iff* ⊢ apply[f, x, v],
(iii) ⊢ apply[**evlis**, {x}, v] *iff* ⊢ evlis[x, v],
(iv) ⊢ apply[**evcon**, {x}, v] *iff* ⊢ evcon[x, v],
(v) ⊢ apply[**subst**, {x, p, b}, v] *iff* ⊢ subst[x, p, b, v],
(vi) ⊢ apply[**point**, {x, w}, v] *iff* ⊢ point[x, w, v]
where ⊢ *means provability in* **Hy**, *and* {· · ·} *may be replaced by* [· · ·] *or* (· · ·).

Proof. Given a tree t, we define its size $|t|$ inductively as follows:
(1) $t = [x] \Rightarrow |t| = 1$,
(2) $t = [x; t_1, \ldots, t_n]\ (n \geq 1) \Rightarrow |t| = 1 + \Sigma|t_i|$

The proof will be carried out by induction o. the size of trees.

(⇒) (i) Suppose $P \vdash$ apply[**eval**, $\{e\}$, v]. We have two cases.

Case 1: $e = [f \cdot x]$. In this case P is of the following form:

```
[apply[eval, {e}, v];
 ...
  [apply[λ([E = [F · X]];
     cond[atom[E]: apply[F, X];
       '1: apply('F, evlis[X])]),{e}, v];
      ...
      [eval[cond[atom[e]: apply[f, x]; '1: apply('f, evlis[x])], v];
       ...
        [evcon[[atom[e]: apply[f, x]; . . .], v];
         ...
          [eval[atom[e], 1]; . . .],
          ...
          [eval[apply[f, x], v]; Q]
        ]
      ...
    ]
  ]
|
```

where

$$Q \vdash \text{apply}[\textbf{apply}, [f, x], v].$$

Since $|Q| < |P|$, by induction hypothesis, we can find an R such that

$$R \vdash \text{apply}[f, x, v].$$

Hence we have

$$[\text{eval}[e, v]; R] \vdash \text{eval}[e, v].$$

Case 2: $e = (f \cdot x)$. In this case, similarly as in Case 1, we can verify that there are subtrees S, T of P and a sexp z such that

$$S \vdash \text{apply}[\textbf{evlis}, [x], z],$$
$$T \vdash \text{apply}[\textbf{apply}, (f, z), v].$$

Since $|S|, |T| < |P|$, by induction hypothesis, we can find U, V such that

$$U \vdash \text{evlis}[x, z],$$
$$V \vdash \text{apply}[f, z, v].$$

Hence, we have

$$[\text{eval}[e, v]; U, V] \vdash \text{eval}[e, v].$$

We have thus proved the only if part of (i). We omit the proof of the remaining parts of the theorem since they can be proved by similar purely combinatorial arguments. □

Corollary 5.5. $e \triangleright v$ *iff* $\text{eval}[e] \triangleright v$.

5.3. Function definition

Although we can define any partial recursive function in **Hy**, function definitions in terms of label expressions or Y-combinator are rather cumbersome. So, for the sake of practical convenience, we introduce the concept of a global environment of function definitions and define a formal system $\mathbf{Hy}_d$ relative to a global environment d. Since a theorem similar to Theorem 5.3 holds for $\mathbf{Hy}_d$, the computational power of $\mathbf{Hy}_d$ is the same as that of **Hy**. We are indebted to Masami Hagiya for the idea of introducing global environments into **Hy**.

We will call, as usual, any sexp d a *global environment*. Let d be a global environment. To define $\mathbf{Hy}_d$, we first modify **Apply** and define $\mathbf{Apply}_d$ as follows. $\mathbf{Apply}_d$ is like **Apply** except that we have the following two axiom schemata in place of **Apply** · 6:

$$\text{atom}[F];\ \text{ne}[F, 1];\ \text{ne}[F, \text{eq}];\ \text{ne}[F, \text{cond}];\ \text{assoc}[F, d, [F \cdot G]];\ \text{apply}[G, X, V] \rightarrow \text{apply}[F, X, V]$$

and

$$\text{atom}[F];\ \text{ne}[F, 1];\ \text{ne}[F, \text{eq}];\ \text{ne}[F, \text{cond}];\ \text{assoc}[F, d, U];\ \text{mole}[U];\ \text{eval}[F, G];\ \text{apply}[G, X, V] \rightarrow \text{apply}[F, X, V]$$

We also put

Assoc =
[→assoc[X, 0, 0],
ne[L, 0]; caar[L, X]; car[L, V]→assoc[X, L, V],
ne[L, 0]; caar[L, K]; ne[X, K]; cdr[L, R]; assoc[X, R, V]→
assoc[X, L, V]].

We then define $\mathbf{Hy}_d$ by putting

$$\mathbf{Hy}_d = \mathbf{Eval} \,\&\, \mathbf{Evlis} \,\&\, \mathbf{Apply}_d \,\&\, \mathbf{Evcon} \,\&\, \mathbf{Subst} \,\&\, \mathbf{Point} \,\&\, \mathbf{Assoc} \,\&\, \mathbf{Eq} \,\&\, \mathbf{Ne} \,\&\, \mathbf{Atom} \,\&\, \mathbf{Mole} \,\&\, \mathbf{C^*r}.$$

We will write $e \triangleright v$ (under d) (read: e evaluates to v under d) if $\vdash \text{eval}[e, v]$ (in $\mathbf{Hy}_d$). $\mathbf{Apply}_d$ · 6 may be paraphrased as follows:

$$f \in \mathbf{A},\ f \neq 1,\ f \neq \text{eq},\ f \neq \text{cond},\ \vdash \text{assoc}[f, d, [f \cdot g]],$$

$$\vdash \text{apply}[g, x, v] \Rightarrow \vdash \text{apply}[f, x, v].$$

Note that f and g above are functionally equivalent.

Example 5.4. Let

```
d = [[assoc ·
        λ([X, L = [L1 = [K] · L2]];
            cond[null[L]: 0;
                 eq[K, X]: 'L1;
                 '1: assoc[X, L2]])],
     [null · λ([X]; eq[X, 0])]].
```

Then we have: assoc[x, l] $\triangleright$ v (under d) iff $\vdash$ assoc[x, l, v] (in **Assoc**).

To simplify the notation for global environments, we introduce the following abbreviation.

Let α_i be a closed unit whose denotation is an atom, χ_i a metalist and ε_i a form such that $\delta_i = \lambda(\chi_i; \varepsilon_i)$ is closed. Then we will let

$$\mathbf{Def}\{\alpha_1\chi_1 = \varepsilon_1; \alpha_2\chi_2 = \varepsilon_2; \ldots; \alpha_n\chi_n = \varepsilon_n\}$$

be an abbreviation of the snoclist

$$[[\alpha_1 \cdot \delta_1], [\alpha_2 \cdot \delta_2], \ldots, [\alpha_n \cdot \delta_n]].$$

Let d be any global environment. We define another global environment D as follows:

```
Def{
  eval[E = [F · X]]
  =cond[atom[E]: apply[F, X];
        '1: apply('F, evlis[X])];

  evlis[X = (X1 · X2)]
  =cond[null[X]: 0;
        '1: cons(eval[X1], evlis[X2])];

  apply[F = (L, P, B), X = [X1, X2]]
  =cond[null[X]: 0;
        atom[F]:
          cond[eq[F, 1]: 'X1;
               eq[F, eq]: eq[X1, X2];
               eq[F, cond]: evcon[X];
               assoc[F, d]: apply(cdr(assoc[F, d]), 'X);
               '1: apply(eval[F], 'X)];
        eq[L, lambda]: eval(subst[X, P, B]);
        eq[L, label]: apply(subst[F, P, B], 'X);
        '1: apply(eval[F], 'X)];

  evcon[X = [(E1, E2) · X2]]
  =cond[null[X]: 0;
```

```
        eval[E1]: eval[E2];
        '1: evcon[X2]];

subst[X, P=[P1 · P2], B=[B1 · B2]]
=cond[null[P]: 'B;
        atom[P]: point[X, P1];
        atom[B]: snoc(subst[X, P1, B1], subst[X, P2, B2]);
        '1: cons(subst[X, P1, B1], subst[X, P2, B2])];

point[X=[X1 · X2], W=(W1 · W2)]
=cond[null[W]: 0;
        atom[W]: 'X;
        null[W2]: point[X1, W1];
        '1: point[X2, W2]];

assoc[X, L=[L1=[K] · L2]]
=cond[null[L]: 0;
        eq[K, X]: 'L1;
        '1: assoc[X, L2]];

atom[X=[X1 · X2]]=eq[X, [X1 · X2]];

null[X]=eq[X, 0];

cons[X, Y]='(X · Y);

snoc[X, Y]='[X · Y];

cdr[[X · Y]]='Y

}
```

We then have the following theorem which can be proved similarly as Theorem 5.4

Theorem 5.6. *Let d and D be as above. Then for any sexps e and v we have*

$$\text{eval}[e] \rhd v \ (\textit{under } D) \ \textit{iff} \ e \rhd v \ (\textit{under } d).$$

Remark. Our proof of Theorem 5.4 and 5.6 is purely combinatorial and constructive. Compare our approach with that of Gordon [4], where he proves the correctness of the universal functions of Pure Lisp by means of denotational semantics.

Acknowledgment

I am greatly influenced, through conversations and publications, by the philosophy of Professor John McCarthy.

Professor Satoru Takasu has constantly encouraged and given suggestions throughout my research on symbolic expressions.

Mr. Masami Hagiya has implemented the first interpreter of Hyperlisp and also contributed many important ideas to the theory.

Professor Shigeru Igarashi gave me helpful suggestions on the first draft of the paper.

I wish to express my sincerest thanks to all of them.

References

[1] J. Allen, *Anatomy of LISP* (McGraw-Hill, New York, 1978).

[2] N. Bourbaki, *Theory of Sets* (Hermann, Paris and Addison-Wesley, Reading, MA, 1968).

[3] H.B. Curry and R. Feys, *Combinatory Logic, Vol.* 1 (North-Holland, Amsterdam, 1968).

[4] M. Gordon, Models of Pure Lisp, Department of Machine Intelligence, Experimental programming reports: No. 30, University of Edinburgh (1973).

[5] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, *Comm. ACM* **3** (1960) 184–195.

[6] J. McCarthy et al., *LISP* 1.5 *Programmer's Manual* (MIT Press, Cambridge, MA, 1962).

[7] J. McCarthy, History of LISP, *SIGPLAN Notices* **13** (1978) 217–224.

[8] W. Quine, *Mathematical Logic* (Harvard, University Press, Cambridge, MA, 1955).

[9] M. Sato, Algebraic structure of symbolic expressions, Technical Report TR82-05, Faculty of Science, University of Tokyo (1982).

[10] M. Sato and M. Hagiya, Hyperlisp, in: J.W. de Bakker and J.C. van Vliet, Eds., *Algorithmic Languages* (North-Holland, Amsterdam, 1981) 251–269.

[11] R. Smullyan, *Theory of Formal Systems*, Annals of Mathematics Studies **47** (Princeton University Press, Princeton, NJ, 1961).

Publ. RIMS, Kyoto Univ.
21 (1985), 455-540

Theory of Symbolic Expressions, II

By

Masahiko SATO*

Abstract

A new domain $\mathbb{S}$ of symbolic expressions is introduced and its structure is studied formally. To study $\mathbb{S}$ formally an intuitionistic first order theory, SA, is introduced. SA is a theory adequate for developing elementary metamathematics within it. Gödel's second incompleteness theorem is proved formally within SA to show the adequacy. A modified version of Post-Smullyan's formal system is used to define basic concepts in SA. The close relation between formal systems and the logic programming language Qute is also pointed out.

Introduction

This paper is a continuation to our former paper Sato [10], which we will refer to as I in the sequel. In this paper we continue our study of the domain $\mathbb{S}$ of symbolic expressions. In I we studied the domain $\mathbb{S}$ informally, but in this paper we treat $\mathbb{S}$ formally within a formal mathematical theory.

Through our attempts at formalization of the domain $\mathbb{S}$ we encountered several technical difficulties. Most of these difficulties came from the fact that cons of 0 and 0 was again 0. (We will not go into the details of the difficulties, but we just mention that they are mostly related to the induction schema on sexps.) We were therefore forced to reconsider the domain itself, and by a simple modification (or, rather simplification) on the definition of symbolic expressions we got a new domain. This domain, which we will denote by the symbol $\mathbb{S}$, will be the objective of our study in this paper. We will refer to our old domain of symbolic expressions which we studied in I as $\mathbb{S}_{old}$.

This paper can be read without any familiarity with I. We should, however, remark that these two domains are very similar to each other and we will

This paper is based on the result of activities of working groups for the Fifth Generation Computer Systems Projects.

Communicated by S. Takasu, April 25, 1984.

* Department of Information Science, Faculty of Science, University of Tokyo, Tokyo, 113, Japan.

study our new domain with the same spirit as in I.

Besides our previous works [10, 11, 12], the domain of symbolic expressions recently attracted the attention of some logicians. Feferman [4] introduced second order theories of symbolic expressions and showed that elementary metamathematics can be naturally developed within his systems. Hayashi [7] also introduced a theory of symbolic expressions and gave sound foundation for his computer implemented system that synthesizes a LISP program from the constructive proof of its specification. The most important reason for the choice of symbolic expressions as the domain of discourse is because they provide a natural and easy way of encoding the metamathematical entities such as proofs or programs. We will adopt the domain of symbolic expressions as our basic objects of our study for the very same reason.

The paper is organized as follows. In Section 1, we introduce our new domain $\mathbb{S}$ of symbolic expressions informally. In Section 2, we introduce the concept of a *formal system*, which is a simplified version of the corresponding concept we studied in I. As in I, formal systems will play fundamental roles in our formal study of $\mathbb{S}$. We also point out that a formal system is essentially equivalent to a program written in a logic programming language. In Section 3 we introduce a formal theory of symbolic expressions which we call $\mathbb{BSA}$ (for *B*asic *S*ymbolic *A*rithmetic). We also explain the intended interpretation of the theory. In Section 4 we introduce a formal system $\mathbb{FOT}$ (for *F*irst *O*rder *T*heory). In $\mathbb{FOT}$ we can regard any sexp as an *axiom system* and we can define arbitrary axiom system with countable first-order language over intuitionistic (or classical) logic by taking suitable sexp for the axiom system.

Formal development of mathematics and metamathematics on the domain $\mathbb{S}$ begins from Section 5. In Section 5 we introduce an axiom system $\mathbb{SA}$ (for *S*ymbolic *A*rithmetic), which is a conservative extension of $\mathbb{BSA}$, as a system which is adequate for the formal development of metamathematics within the system.

In Section 6 we develop simple mathematics within $\mathbb{SA}$ as a preparation for Section 7 where we develop elementary metamathematics within $\mathbb{SA}$. Section 8 will be devoted to the formal proof of some of Gödel's incompleteness theorems.

§ 1. Symbolic Expressions

1.1. sexps

We define symbolic expressions (*sexps*, for short) by the following inductive clauses:

1. $*$ is a sexp.
2. If s and t are sexps then $cons(s, t)$ is a sexp.
3. If s and t are sexps then $snoc(s, t)$ is a sexp.

All the sexps are constructed by finitely many applications of the above three clauses, and sexps constructed differently are distinct. We denote the set of all the sexps by $\mathbb{S}$. We denote the image of the function *cons* by $\mathbb{M}$ and that of *snoc* by $\mathbb{A}$. We then have two bijective functions:

$$cons: \mathbb{S}\times\mathbb{S}\to\mathbb{M}$$
$$snoc: \mathbb{S}\times\mathbb{S}\to\mathbb{A}$$

Moreover, by the construction of $\mathbb{S}$, we see that $\mathbb{S}$ is the union of three mutually disjoint sets $\{*\}$, $\mathbb{M}$ and $\mathbb{A}$. In other words, $\mathbb{S}$ satisfies the following domain equation:

$$\mathbb{S}\equiv\{*\}+\mathbb{A}+\mathbb{M}\cong\{*\}+\mathbb{S}\times\mathbb{S}+\mathbb{S}\times\mathbb{S}$$

We will use the symbol '$\equiv$' as informal equality symbol, and will reserve the symbol '$=$' for the formal equality sign. Elements in $\mathbb{M}$ are called *molecules* and those in $\mathbb{A}$ are called *atoms* and $*$ is called *nil*. We define two total functions, *car* and *cdr*, on $\mathbb{M}$ by the equations:

$$car(cons(s, t))\equiv s$$
$$cdr(cons(s, t))\equiv t$$

Similarly we define two total functions, *cbr* and *ccr*, on $\mathbb{A}$ by the equations:

$$cbr(snoc(s, t))\equiv s$$
$$ccr(snoc(s, t))\equiv t$$

Compositions of the functions *car*, *cbr*, *ccr* and *cdr* will be abbreviated following the convention in LISP. For instance:

$$cabcdr(t)\equiv car(cbr(ccr(cdr(t))))$$

We must introduce some notations for sexp. The so-called dot notation and list notation introduced below is fundamental.

$$[.\ t]\equiv t$$

$$[t_1,\ldots, t_n . t_{n+1}] \equiv cons(t_1, [t_2,\ldots, t_n . t_{n+1}]) \ (n \geqq 1)$$
$$[t_1,\ldots, t_n] \equiv [t_1,\ldots, t_n . *] \ (n \geqq 0)$$

In particular we have

$$[s . t] \equiv cons(s, t)$$
$$[\ ] \equiv *$$

A sexp of the form $[t_1,\ldots, t_n]$ will be called a *list*. We will also use the following abbreviations.

$s[\,.\,t]$ for $[s . t]$
$s[t_1,\ldots, t_n . t_{n+1}]$ for $[s, t_1,\ldots, t_n . t_{n+1}]$
$s[t_1,\ldots, t_n]$ for $[s, t_1,\ldots, t_n]$

For *snoc*, we only use the following notation

$$(s . t) \equiv snoc(s, t)$$

Parentheses will also be used for grouping. Thus (t) will *not* denote $snoc(t, *)$ but will denote t. (Readers of our previous papers should note the change of notations.)

The basic *induction schema* on $\mathbb{S}$ can be stated as follows. Let $\Phi(t)$ be a proposition about a sexp t. Then we may conclude that $\Phi(t)$ holds for any t, if we can prove the following three propositions.

(i) $\Phi(*)$
(ii) If $\Phi(s)$ and $\Phi(t)$ then $\Phi([s . t])$
(iii) If $\Phi(s)$ and $\Phi(t)$ then $\Phi((s . t))$

1.2. symbols and variables

An atom of the form

$$(* . x)$$

will be called a *symbol*. Let Σ be the set of 128 ASCII characters. We define an injective function $\rho: \Sigma \to \mathbb{M}$ by using 7 bit ASCII codes, regarding $*$ as 0 and $[*]$ as 1. For instance, we have

$$\rho(a) \equiv [[*], [*], *, *, *, *, [*]]$$
$$\rho(1) \equiv [*, [*], [*], *, *, *, [*]]$$

We extend ρ homomorphically to Σ^*, i.e., we define $\rho^*: \Sigma^* \to \mathbb{M}$ by $\rho^*(\sigma_1 \cdots \sigma_k) \equiv [\rho(\sigma_1),\ldots, \rho(\sigma_k)]$ $(\sigma_i \in \Sigma)$. Now consider a string of alphanumeric characters such that

(i) its length is longer than 1,

(ii) it begins with a lowercase character and

(iii) its second character is a non-numeric character.

Such a string will be called a *name*. Let π be a name. Then, by definition, π denotes the symbol

$$(*.[*.\rho^*(\pi)])$$

An atom of the form

$$(\mathrm{var}.x)$$

is called a *variable*. (Note that 'var' denotes a specific symbol. See Example 1.1 below.) We introduce notations for variables. A string of alphanumeric characters such that

(i) it begins with an uppercase character, or

(ii) it consists of a single lowercase character, or

(iii) its first character is lowercase and its second character is a numeral

denotes a variable as follows. Let π be such a string. Then, by definition, π denotes the variable

$$(\mathrm{var}.(*.[*.\rho^*(\pi)]))$$

We will regard the under score character '_' as a lower case character for convenience.

Example 1.1.

$$\mathrm{var} \equiv (*.[*.[[[*], [*], [*], *, [*], [*], *], [[*], [*], *, *, *, *, [*]], [[*], [*], [*], *, *, [*], *]]])$$

$$\mathrm{Var} \equiv (\mathrm{var}.(*.[*.[[[*], *, [*], *, [*], [*], *], [[*], [*], *, *, *, *, [*]], [[*], [*], [*], *, *, [*], *]]])) \quad \square$$

§2. Formal Systems

2.1. formal system

In I, we have defined the concept of a *formal system*. Here we will redefine a formal system by giving a simpler definition of it. As explained in I, our concept of a formal system has its origin in Smullyan [13]. However, unlike Smullyan's, our formal system will be defined directly as a sexp. This has the advantage of making the definition of a universal formal system simpler. Another practically very important aspect of our concept of a formal system is that

it can be quite naturally viewed as a so-called logic program. This means that we can execute formal systems on a computer if we have an interpreter for them. In fact, Takafumi Sakurai of the University of Tokyo implemented such an interpreter. (See [12].) We can therefore use formal systems both as theoretically and practically basic tools for our study of symbolic expressions.

Note. When we introduced formal systems in I, we were ignorant of the programming language PROLOG. But after we had submitted I for publication, we knew the existence of the language. Since it was clear, for any one who knows both PROLOG and Post-Smullyan's formal system (or, the concept of inductive definition), that they are essentially the same, we asked T. Sakurai to implement an interpreter for our formal systems which we introduced in I. The interpreter was named Hyperprolog, and it was used to debug the definition of **Ref** which we gave in I. In this way we could correct bugs in our formal systems in the stage of proof reading. We believe that the existence of an interpreter is essential for finding and correcting such *bugs*. We also remark that Hyperprolog was quite useful in designing our new formal system, which we are about to explain, since it can be simulated on Hyperprolog. Finally we remark that we have designed a new programming language called Qute which can compute relations defined by our new formal system. Qute was also implemented by T. Sakurai. (See Sato and Sakurai [12].) □

Now let us define our formal system. We will call, by definition, any sexp a *formal system*. Our objective, then, is to define a relation *proves*(p, a, FS) which holds among certain triples p, a, FS of sexps where the sexp FS is treated as a formal system. We will employ informal inductive definitions to define the relation *proves*. We will say that p is a *proof* of a in the *formal system* FS, if *proves*(p, a, FS) holds. We write:

$$p \vdash_{FS} a \text{ for } \textit{proves}\,(p, a, FS)$$

We will say that a is a *theorem* in FS if *proves* (p, a, FS) holds for some p, and will use the notation:

$$\vdash_{FS} a$$

for it.

2.2 inductive definitions

As an example of informal inductive definition, let us define the relation *member* (x, L) which means that x is a member of L:

(M1) $\Longrightarrow member(x, [x.\ L])$

(M2) $member(x, L) \Longrightarrow member(x, [y.\ L])$

(M1) means that the relation $member(x, [x.\ L])$ holds unconditionally for any sexp x and L, and (M2) says that if the relation $member(x, L)$ holds then the relation $member(x, [y.\ L])$ also holds for any sexp x, L and y. We have omitted the usual extremal clause which states that the relation $member(x, L)$ holds only when it can be shown to be so by finitely many applications of the clauses (M1) and (M2).

Let us now consider about the nature of (informal) inductive definitions in general. All inductive definitions which we consider in this paper consist of a finite set of clauses (or, rules) of the form

(Γ) $\gamma_1,\ldots,\gamma_n \Longrightarrow \gamma$

where $n \geqq 0$ and Γ is the name of the clause which is used to identify the clause. For example, in (M1) n is 0 and in (M2) n is 1. Suppose we have a finite set of inductive clauses like above, and we could conclude that a certain specific relation holds among specific sexps from these inductive clauses. Let us write our conclusion as α. (If our set of inducitve clauses consist only of (M1) and (M2) above, then α is of the form $member(x, L)$ where x and L are certain specific sexps such as orange or [apple, orange].) We now show that we can associate with α an informal proof Π of α. According to the extremal clause, α is obtained by applying our inductive clauses finitely many times. Let (Γ) be the last applied clause. Since the clause (Γ) is schematic, when we apply (Γ) we must also specify for each schematic variables x_i a sexp v_i as its value. Let $x_1,\ldots, x_k$ be an enumeration of schematic variables occurring in (Γ) and let

$$\Delta \equiv \langle x_1 := v_1,\ldots, x_k := v_k \rangle$$

By substituting v_i for x_i, we can obtain the following instance of (Γ):

(Γ_Δ) $\alpha_1,\ldots, \alpha_n \Longrightarrow \alpha$

Note that the conclusion of (Γ_Δ) must be α by our assumption that α is obtained by applying (an instance of) (Γ). That (Γ_Δ) is applicable also means that each α_i has already been shown to hold by applying inductive clauses finitely many times. Since the number of applications of inductive clauses which was used to show α_i is smaller than that was required to show α, we may assume, as induction hypothesis, that we have an informal proof Π_i of α_i for each $1 \leqq i \leqq n$. Using

these data, we can construct a proof Π of α as the figure of the form:

$$\frac{\Pi_1 \cdots \Pi_n}{(\Gamma)\Delta}$$

Example 2.1.

From (M1) and (M2), we can conclude that *member*(orange, [apple, orange]) holds, and we have the following proof associated with this.

$$\frac{\overline{(\mathrm{M1})\langle x := \mathrm{orange},\ L := [\ ]\rangle}}{(\mathrm{M2})\langle x := \mathrm{orange},\ y := \mathrm{apple},\ L := [\mathrm{orange}]\rangle}$$ □

2.3. definition of the relation proves

Based on this intuitive idea of informal *proof*, we define the relation *proves* etc. as follows. First we define *ne* (for *n*ot *e*qual) which has the property that $ne(x, y)$ holds iff x and y are two distinct sexps.

(N1) $\Longrightarrow ne(*, [u\,.\,v])$
(N2) $\Longrightarrow ne(*, (u\,.\,v))$
(N3) $\Longrightarrow ne([s\,.\,t], *)$
(N4) $\Longrightarrow ne((s\,.\,t), *)$
(N5) $\Longrightarrow ne([s\,.\,t], (u\,.\,v))$
(N6) $\Longrightarrow ne((s\,.\,t), [u\,.\,v])$
(N7) $ne(s, u) \Longrightarrow ne([s\,.\,t], [u\,.\,v])$
(N8) $ne(t, v) \Longrightarrow ne([s\,.\,t], [u\,.\,v])$
(N9) $ne(s, u) \Longrightarrow ne((s\,.\,t), (u\,.\,v))$
(N10) $ne(t, v) \Longrightarrow ne((s\,.\,t), (u\,.\,v))$

We next define *assoc* which is used to get the value of a variable from a given environment.

(A1) $\Longrightarrow assoc(x, [[x\,.\,v]\,.\,L], v)$
(A2) $ne(x, y), assoc(x, L, v) \Longrightarrow assoc(x, [[y\,.\,w]\,.\,L], v)$.

Example 2.2.

assoc(c, [[a . apple], [b . banana], [c . carrot]], carrot) □

The relation *get* is used to extract the i-th member of a list L.

(G1) $\Longrightarrow get(*, [v\,.\,L], v)$
(G2) $get(i, L, v) \Longrightarrow get([*\,.\,i], [w\,.\,L], v)$

Example 2.3.

get([*, *], [lisp, prolog, qute], qute) □

The following relation *eval* gives a simple evaluator of a sexp under a certain environment. Substitution of values to variables can be simulated by *eval*.

(E1) $assoc((\mathrm{var}\,.\,t), Env, v) \Longrightarrow eval((\mathrm{var}\,.\,t), Env, v)$

(E2) $\Longrightarrow eval(*, Env, *)$

(E3) $eval(s, Env, u), eval(t, Env, v) \Longrightarrow eval([s\,.\,t], Env, [u\,.\,v])$

(E4) $eval(s, Env, u), eval(t, Env, v) \Longrightarrow eval((\mathrm{snoc}\,.\,[s, t]), Env, (u\,.\,v))$

(E5) $\Longrightarrow eval((*\,.\,t), Env, (*\,.\,t))$

(E6) $\Longrightarrow eval((\mathrm{quote}\,.\,t), Env, t)$

We will use the following abbreviations for atoms whose *cbr* is snoc or quote.

$(:\,s\,.\,t)$ for $(\mathrm{snoc}\,.\,[s, t])$

$(:\,t)$ for $(\mathrm{snoc}.\,[t, *])$

$'t$ for $(\mathrm{quote}\,.\,t)$

Example 2.4.

$eval([x, \mathrm{of}, \mathrm{y}, \mathrm{and}, \mathrm{z}, \mathrm{is}, '(\mathrm{apple}\,.\,\mathrm{orange})],$
$[[\mathrm{x}\,.\,\mathrm{snoc}], [\mathrm{y}\,.\,\mathrm{apple}], [\mathrm{z}\,.\,\mathrm{orange}]],$
$[\mathrm{snoc}, \mathrm{of}, \mathrm{apple}, \mathrm{and}, \mathrm{orange}, \mathrm{is}, (\mathrm{apple}\,.\,\mathrm{orange})])$ □

In terms of these relations we can now define *proves* ans *lproves*.

(L1) $\Longrightarrow lproves([\,], [\,], FS)$

(L2) $proves(p, a, FS), lproves(P, A, FS)$
$\Longrightarrow lproves([p\,.\,P], [a\,.\,A], FS)$

(P1) $assoc(Prd, FS, R), get(i, R, [c\,.\,C]), eval(c, Env, a),$
$eval(C, Env, A), lproves(P, A, FS)$
$\Longrightarrow proves([[Prd, i, Env]\,.\,P], [Prd\,.\,a], FS)$

We can also define the relation $\vdash_{FS} a$ by the following inductive definition.

(T1) $proves(p, a, FS) \Longrightarrow theorem(a, FS)$

We show by an example how our intuitive idea of proof has been formalized. Recall that the relation *assoc* was defined by the two clauses (A1) and (A2) and that its definition depends also on the relation *ne*. Since *ne* has 10 clauses ((N1)–(N10)), we need 12 clauses to define *assoc*. We formalize these 12 clauses in two steps. In the first step we formalize clauses (A1) and (A2) into a sexp *Assoc* and clauses (N1)–(N10) into a sexp *Ne*. In the second step we obtain a formal sytem [*Assoc*, *Ne*] as a formalization of *assoc* and *ne*. The sexp *Assoc*, which is the translation of clauses (A1) and (A2), is defined as follows:

```
[ assoc
      , [[x, [[x . v] . L], v]]
      , [[x, [[y . w] . L], v]
            , ne[x, y]
            , assoc[x, L, v]]
      ]
```

We explain the general mechanism of our translation of clauses. We translate clauses that are used to define a same relation into a single sexp. We therefore translate (A1) and (A2) into *Assoc* and (N1)–(N10) into *Ne*. Recall that each clause is of the form:

$$\gamma_1, \ldots, \gamma_n \Longrightarrow \gamma$$

and that the general form of γ or γ_i is:

$$\mathbb{P}rd(Arg_1, \ldots, Arg_k)$$

We translate $\mathbb{P}rd$ into corresponding symbol. For instance *assoc* is translated into 'assoc'. Arg's are translated as follows. Since Arg is a schematic expression for sexp it has one of the following forms: (i) a schematic variable, (ii) $*$, (iii) $[\alpha . \beta]$, (iv) $(\alpha . \beta)$. In case of (i) we translate it into corresponding (formal) variable. Thus x is translated into 'x'. If Arg is $*$ then it is translation into $*$. If Arg is of the form (iii), its translation is $[\alpha^* . \beta^*]$ where α^* (β^*) is the translation of α (β, resp.). Similarly, but slightly differently, case (iv) is translated into $(: \alpha^* . \beta^*)$ if α in not $*$ and it is translated into itself if α is $*$. (Since the translation must be one to one, we cannot translate $(\alpha . \beta)$ into $(\alpha^* . \beta^*)$ because, then, (ii)–(iv) will leave no room for the translation of schematic variables.) By extending this translation naturally we obtain the above translation of (A1) and (A2). For the sake of readability we introduce the following abbreviation for the above sexp *Assoc*.

```
+ assoc
      | x, [[x . v] . L], v
      | x, [[y . w] . L], v
            – ne[x, y]
            – assoc[x, L, v]
```

Example 2.5. By the similar idea as above we can translate the informal proof in Example 2.1 into the following formal proof p:

[[member, [*], [[x . orange], [y . apple], [𝕃 . [orange]]]],
[[member, *, [[x . orange], [𝕃 . [ ]]]]]]

Let *Member* be the following sexp:

```
+ member
    | x, [x . 𝕃]
    | x, [y . 𝕃]
          - member[x, 𝕃]
    ;
```

Then we can easily verify that

$$p \vdash_{[Member]} \text{member[orange, [apple, orange]]}$$

holds and hence

$$\vdash_{[Member]} \text{member[orange, [apple, orange]]}$$

holds. □

2.4. universal formal system

By translating the relations we have defined so far we obtain a formal system 𝕌niv which is universal among all the formal systems. We thus define 𝕌niv as the sexp:

$$\mathbb{U}\text{niv} \equiv [Ne, Assoc, Get, Eval, Lproves, Proves, Theorem]$$

where *Ne*, *Assoc*, *Get*, *Eval*, *Lproves*, *Proves* and *Theorem* are respectively:

```
+ ne
    | *, [u . v]
    | *, (: u . v)
    | [s . t], *
    | (: s . t), *
    | [s . t], (: u . v)
    | (: s . t), [u . v]
    | [s . t], [u . v]
          - ne[s, u]
    | [s . t], [u . v]
          - ne[t, v]
    | (: s . t), (: u . v)
          - ne[s, u]
```

```
        | (: s . t), (: u . v)
                - ne[t, v]
        ;
+ assoc
        | x, [[x . v] . L], v
        | x, [[y . w] . L], v
                - ne[x, y]
                - assoc[x, L, v]
        ;
+ get
        | *, [v . L], v
        | [* . i], [w . L], v
                - get[i, L, v]
        ;
+ eval
        | (: var . t), Env, v
                - assoc[(: var . t), Env, v]
        | *, Env, *
        | [s . t], Env, [u . v]
                - eval[s, Env, u]
                - eval[t, Env, v]
        | (: snoc . [s, t]), Env, (: u . v)
                - eval[s, Env, u]
                - eval[t, Env, v]
        | (: * . t), Env, (: * . t)
        | (: quote . t), Env, t
        ;
+ lproves
        | [ ], [ ], FS
        | [p . P,], [a . A], FS
                - proves[p, a, FS]
                - lproves[P, A, FS]
        ;
+ proves
        | [[Prd, i, Env] . P], [Prd . a], FS
                - assoc[Prd, FS, R]
```

```
                - get[i, R, [c.C]]
                - eval[c, Env, a]
                - eval[C, Env, A]
                - lproves[P, A, FS]
        ;
+ theorem
        | a, FS
                - proves[p, a, FS]
        ;
```

The following theorem establishes that Univ is in fact a universal formal system.

Theorem 2.1.

(i) $ne(x, y) \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{ne}[x, y]$

(ii) $assoc(x, L, v) \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{assoc}[x, L, v]$

(iii) $get(i, L, v) \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{get}[i, L, v]$

(iv) $eval(t, E, v) \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{eval}[t, E, v]$

(v) $lproves(P, A, FS) \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{lproves}[P, A, FS]$

(vi) $proves(p, a, FS) \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{proves}[p, a, FS]$

(vii) $theorem(a, FS) \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{theorem}[a, FS]$

We omit the simple but tedious combinational proof of this theorem. The following corollary is simply a restatement of the last two sentences of this theorem.

Corollary 2.2.

(i) $p \vdash_{FS} a \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{proves}\,[p, a, FS]$

(ii) $\vdash_{FS} a \Longleftrightarrow \vdash_{\mathbf{Univ}} \mathrm{theorem}\,[a, FS]$

§3. Formal Theory of Symbolic Expressions: BSA

In this section we introduce a formal theory of symbolic expressions which we call BSA (for *B*asic *S*ymbolic *A*rithmetic). The theory is a first order intuitionistic theory which is proof theoretically equivalent to HA (Heyting arithmetic).

Traditionally, metamathematical entities such as *terms*, *wffs* and *proofs* have been considered as concrete figures which can be displayed on a sheet of paper (with some kind of abstraction which is necessary so as to allow finite but arbitrarily large figures). Our standpoint is, however, not like this but to

regard these entities as symbolic expressions. By taking this standpoint we can define SA formally in terms of a formal system. It is also possible to define BSA in this way, but for the convenience of the reader who is perhaps so accustomed to the traditional approach we first define BSA in the usual way and will then explain how BSA so defined can be isomorphically translated into S. We reserve BSA as the name for the system which we will define as a formal system in Section 3.7, and use $\mathcal{BSA}$ to denote the theory which we now define by a traditional method.

3.1. language

The language of $\mathcal{BSA}$ consists of the following symbols.

- *individual symbols*: **nil**
- *function symbols*: **cons**, **snoc**
- *pure variables*: $\mathbf{var}_t$ for each sexp t
- *predicate symbols*: **eq** (equal), **lt** (less than)
- *logical symbols*: **and**, **or**, **imply**, **all**, **exist**
- *other symbols*: (,), ' , ' (comma), **free**

3.2. variables, terms and wffs

Using the language introduced above, we define syntactic entities of $\mathcal{BSA}$. We first define *variables* as follows.

1. For each sexp t, the pure variable $\mathbf{var}_t$ is a variable.
2. If x is a variable then $\mathbf{free}(x)$ is a variable.

For a variable x we define its *pure part* as follows.

1. If x is a pure variable then its pure part is x itself.
2. If the pure part of x is y then the pure part of $free\ (x)$ is also y.

The definition of *terms* is as follows.

1. A variable is a term.
2. **nil** is a term.

3–4. If s and t are terms then $\mathbf{cons}(s, t)$ and $\mathbf{snoc}(s, t)$ are terms.

We define *wffs* (well formed formulas) as follows.

1–2. If s and t are terms then $\mathbf{eq}(s, t)$ and $\mathbf{lt}(s, t)$ are wffs.

3–4. If $a_1,\ldots, a_n$ $(n \geqq 0)$ are wffs then $\mathbf{and}(a_1,\ldots, a_n)$ and $\mathbf{or}(a_1,\ldots, a_n)$ are wffs.

5. If $a_1,\ldots, a_n$ $(n \geqq 0)$ and b are wffs then $\mathbf{imply}((a_1,\ldots a_n), b)$ is a wff.

6–7. If $x_1,\ldots, x_n$ $(n \geqq 0)$ are distinct pure variables and a is a wff then $\mathbf{all}((x_1,\ldots, x_n), a)$ and $\mathbf{exist}((x_1,\ldots x_n), a)$ are wffs.

A wff is called an *atomic wff* if it is constructed by the clauses 1–2 above, and a wff is called a *quantifier free wff* if it is constructed by the clauses 1–5 above. We will call both a term and a wff as a *designator*.

We will use the following symbols with or without subscripts as syntactic variables for specific syntactic objects.

x, y, z for variables
r, s, t, u, v for terms
a, b, c for wffs
d, e for designators

3.3. abbreviations

We introduce the folowing abbreviations.

$\# x$ for $\mathbf{free}(x)$
$s = t$ for $\mathbf{eq}(s, t)$
$s < t$ for $\mathbf{lt}(s, t)$
$s \leqq t$ for $\mathbf{or}(\mathbf{lt}(s, t), \mathbf{eq}(s, t))$
$a_1 \wedge \cdots \wedge a_n$ for $\mathbf{and}(a_1,\ldots, a_n)$
$a_1 \vee \cdots \vee a_n$ for $\mathbf{or}(a_1,\ldots, a_n)$
$a_1,\ldots, a_n \rightarrow b$ for $\mathbf{imply}((a_1,\ldots, a_n), b)$
$a \leftrightarrow b$ for $\mathbf{and}(\mathbf{imply}((a), b), \mathbf{imply}((b), a))$
$\forall(x_1,\ldots, x_n; a)$ for $\mathbf{all}((x_1,\ldots, x_n), a)$
$\exists(x_1,\ldots, x_n; a)$ for $\mathbf{exist}((x_1,\ldots, x_n), a)$

We assume that the binding power of the operators $\wedge$, $\vee$ and $\rightarrow$ decrease in this order, and we insert parentheses when necessary to insure unambiguous reading.

3.4. substitutions and free variables

Let t be a term, x be a variable and d be a designator. We then define a designator e which we call the result of *substituting* t *for* x *in* d as follows. The definition requires one auxiliary concept, namely, the *elevation of* a term *with* respect to a finite sequence of pure variables, which we also define below.

I.1.1. If d is x then e is t.
I.1.2. If d is a variable other than x then e is d.

I.2. If d is **nil** then e is **nil**.

I.3. If d is **cons**(t_1, t_2) and e_1 (e_2) is the results of substituting t for x in t_1 (t_2, resp.) then e is **cons**(e_1, e_2).

I.4. If d is **snoc**(t_1, t_2) and e_1 (e_2) is the result of substituting t for x in t_1 (t_2, resp.) then e is **snoc**(e_1, e_2).

II.1. If d is **eq**(t_1, t_2) and e_1 (e_2) is the result of substituting t for x in t_1 (t_2, resp.) then e is **eq**(e_1, e_2).

II.2. If d is **lt**(t_1, t_2) and e_1 (e_2) is the result of substituting t for x in t_1 (t_2, resp.) then e is **lt**(e_1, e_2).

II.3–4. If d is **and**(a_1,..., **or**) (**or**(a_1,..., a_n)) and e_i ($1 \leqq i \leqq n$) is the result of substituting t for x in a_i then e is **and**(e_1,... e_n) (**or**(e_1,..., e_n), resp.).

II.5. If d is **imply**((a_1,..., a_n), b), e_i ($1 \leqq i \leqq n$) is the result of substituting t for x in a_i and c is the result of substituting t for x in b then e is **imply**((e_1,..., e_n), c).

II.6. If d is **all**((x_1,..., x_n), a), u (y) is the elevation of t (x, resp.) with respect to the sequence of pure variables x_1,..., x_n and b is the result of substituting u for y in a then e is **all**((x_1,..., x_n), b)

II.7. If d is **exist**((x_1,..., x_n), a), u (y) is the elevation of t (x, resp.) with respect to the sequence of pure variables x_1,..., x_n and b is the result of substituting u for y in a then e is **exist** ((x_1,..., x_n, b)

Let t be a term and x_1,..., x_n ($n \geqq 0$) be a sequence of distinct pure variables. We define a term u which we call the elevation of t with respect to x_1,..., x_n as follows.

1.1. If t is a variable whose pure part is x_i for some i ($1 \leqq i \leqq n$) then u is **free**(t).

1.2. If t is a variable whose pure part does not appear in the sequence x_1,..., x_n then u is t.

2. If t is **nil** then u is **nil**.

3. If t is a term **cons**(t_1, t_2) and u_1 (u_2) is the elevation of t_1 (t_2, resp.) with respect to the sequence x_1,..., x_n then u is **cons**(u_1, u_2).

4. If t is a term **snoc**(t_1, t_2) and u_1 (u_2) is the elevation of t_1 (t_2, resp.) with respect to the sequence x_1,..., x_n then u is **snoc**(u_1, u_2).

That the result of substituting a term for a variable in a designator is again a designator of the same type can be proved easily by induction. (To prove this, one must also prove that the elevation of a term with respect to a sequence of distinct pure variables is also a term.)

Example 3.1.

(i) Let x and y be distinct pure variables and let $\mathfrak{a}$ be the wff $\exists(x; x=y)$. Let us substitute x for y in $\mathfrak{a}$. To do so, we must first compute the elevations of x and y with respect to x. They are $\#x$ and y respectively. Now the result of substituting $\#x$ for y in $x=y$ is $x=\#x$. Thus we have that $\exists(x; x=\#x)$ is the result of substituting x for y in $\mathfrak{a}$. Let us call this wff $\mathfrak{b}$. Then the reader should verify that the result of substituting y for x in $\mathfrak{b}$ is $\mathfrak{a}$.

(ii) Let z be a variable distinct from x and y above and consider the wff $\exists(x, y; z=\mathbb{cons}(x, y))$. Then the result of substituting the term $\mathbb{cons}(x, y)$ for z in this wff is calculated similarly as above and we obtain the wff $\exists(x, y; \mathbb{cons}(\#x, \#y)=\mathbb{cons}(x, y))$. □

Remark. As can be seen in the above examples we have avoided the problem of the collision of variables by introducing a systematic way of referring to a *non-local* variable that happens to have the same name as one of the *local* variables. We remark that our method is a generalization of the method due to de Bruijn [3]. □

We can define simultaneous substitution similarly. Let $\mathfrak{t}_1,\ldots,\mathfrak{t}_n$ be a sequence of terms, $x_1,\ldots,x_n$ be a sequence of distinct variables and let $\mathfrak{d}$ be a designator. We will use the notation $\mathfrak{d}_{x_1,\ldots,x_n}[\mathfrak{t}_1,\ldots,\mathfrak{t}_n]$ to denote the result of simultaneously substituting $\mathfrak{t}_1,\ldots,\mathfrak{t}_n$ for $x_1,\ldots,x_n$ in $\mathfrak{d}$.

We say that a variable x *occurs free in* a designator $\mathfrak{d}$ if $\mathfrak{d}_x[\mathbb{nil}]$ is distinct from $\mathfrak{d}$. A designator is said to be *closed* if no variables occur free in it.

We need the following concept in the definition of proofs below. Let $\mathfrak{t}$ be a term, x be a variable and $\mathfrak{d}$ be a designator. We then define a designator $\mathfrak{e}$ which we call the result of *bind substituting* $\mathfrak{t}$ *for* x *in* $\mathfrak{d}$ as follows. The definition goes completely in parallel with the definition of substitution except for the clause I.1.2. We therefore only give the clause I.1.2 below.

I.1.2. If $\mathfrak{d}$ is a variable other than x then:
  if the pure parts of $\mathfrak{d}$ and x are the same then:
    if $\mathfrak{d}$ is a pure variable then $\mathfrak{e}$ is $\mathfrak{d}$;
    if x is a pure variable then $\mathfrak{e}$ is defined so that $\mathfrak{d}\equiv\#\mathfrak{e}$;

if $x \equiv \sharp x_1$ and $d \equiv \sharp d_1$ then e is $\sharp e_1$ where e_1 is the result of bind substituting t for x_1 in d_1;

if the pure parts of d and x are distinct then e is d.

Let $t_1,\ldots, t_n$ be a sequence of terms, $x_1,\ldots, x_n$ be a sequence of variables whose pure parts are distinct and d be a designator. We can define the result of simultaneously bind substituting $t_1,\ldots, t_n$ for $x_1,\ldots, x_n$ in d similarly as above, and we use the notation $d_{x_1,\ldots,x_n}[\![t_1,\ldots, t_n]\!]$ for it.

3.5. proofs

We formulate our fromal theory *BSA* in natural deduction style. Since we eventually give a precise definition of BSA using a formal system, we give here an informal definition in terms of schematic inference rules. Namely an inference rule is a figure of the form:

$$\frac{a_1 \ldots a_n}{a} \qquad n \geqq 0$$

where a_i, a are formulas. a_i may have *assumptions* that are *discharged* at this inference rule, and we show such assumptions by enclosing them by brackets. We call $a_1,\ldots, a_n$ the *premises* and a the *consequence* of the inference rule. We first collect logical rules. The logic we use is the first order intuitionistic logic with equality.

$$(\wedge I)\ \frac{a_1 \ldots a_n}{a_1 \wedge \cdots \wedge a_n} \qquad (\wedge E)_i\ \frac{a_1 \wedge \cdots \wedge a_n}{a_i}\ 1 \leqq i \leqq n$$

$$(\vee I)_i\ \frac{a_i}{a_1 \vee \cdots \vee a_n}\ 1 \leqq i \leqq n \qquad (\vee E)\ \frac{a_1 \vee \cdots \vee a_n \quad \begin{matrix}[a_1]\cdots[a_n] \\ c\end{matrix} \quad \begin{matrix}[a_1]\cdots[a_n] \\ c\end{matrix}}{c}$$

$$(\to I)\ \frac{\begin{matrix}[a_1,\ldots, a_n] \\ b\end{matrix}}{a_1,\ldots, a_n \to b} \qquad (\to E)\ \frac{a_1,\ldots, a_n \to b \quad a_1 \ldots a_n}{b}$$

$$(\forall I)\ \frac{a_{x_1,\ldots,x_n}[\![y_1,\ldots, y_n]\!]}{\forall(x_1,\ldots, x_n; a)} \qquad (\forall E)\ \frac{\forall(x_1,\ldots, x_n; a)}{a_{x_1,\ldots,x_n}[\![t_1,\ldots, t_n]\!]}$$

$$(\exists I)\ \frac{a_{x_1,\ldots,x_n}[\![t_1,\ldots, t_n]\!]}{\exists(x_1,\ldots, x_n; a)} \qquad (\exists E)\ \frac{\exists(x_1,\ldots, x_n; a) \quad \begin{matrix}[a_{x_1,\ldots,x_n}[\![y_1,\ldots, y_n]\!]] \\ b\end{matrix}}{b}$$

$$(=)\ \frac{}{t=t} \qquad (=subst)\ \frac{a_{x_1,\ldots,x_n}[s_1,\ldots, s_n]\ s_1=t_1 \ldots s_n=t_n}{a_{x_1,\ldots,x_n}[t_1,\ldots, t_n]}$$

In the above rules the variables $x_1,\ldots, x_n$ must be distinct pure variables. The

variables $y_1,\ldots, y_n$ must be distinct and must satisfy the *eigen variables conditions*. That is, in $(\forall I)$, they must not occur free in $\forall(x_1,\ldots x_n; \mathscr{a})$ or in any assumption on which $\mathscr{a}_{x_1,\ldots,x_n}[\![y_1,\ldots, y_n]\!]$ depends, and in $(\exists E)$, they must not occur free in $\exists(x_1,\ldots, x_n; \mathscr{a})$, $\mathscr{b}$ or any assumption other than $\mathscr{a}_{x_1,\ldots,x_n}[\![y_1,\ldots, y_n]\!]$ on which the premise $\mathscr{b}$ depends.

Note that we may regard the wffs $\mathfrak{and}(\ )$ and $\mathfrak{or}(\ )$ as representing the truth values *true* and *false* respectively by letting n to be 0 in $(\wedge I)$ and $(\vee E)$. For this reason, we will use $\bot$ as an abbreviation for $\mathfrak{or}(\ )$, $\mathscr{a}\ \neg$ for $\mathscr{a}\to\bot$ and $s\neq t$ for $\neg(s=t)$.

The remaining rules are specific to the theory $\mathbb{BSA}$. First we consider the rules for equality.

$$(cons\neq nil)\ \frac{\mathfrak{cons}(s,\ t)=\mathfrak{nil}}{\bot}\qquad (snoc\neq nil)\ \frac{\mathfrak{snoc}(s,\ t)=\mathfrak{nil}}{\bot}$$

$$(cons\neq snoc)\ \frac{\mathfrak{cons}(s,\ t)=\mathfrak{snoc}(u,\ v)}{\bot}$$

$$(cons=cons)_i\ \frac{\mathfrak{cons}(s_1,\ s_2)=\mathfrak{cons}(t_1,\ t_2)}{s_i=t_i}\quad i=1,\ 2$$

$$(snoc=snoc)_i\ \frac{\mathfrak{snoc}(s_1,\ s_2)=\mathfrak{snoc}(t_1,\ t_2)}{s_i=t_i}\quad i=1,\ 2$$

Next we collect rules for $<$ (leass than).

$$(<*)\ \frac{r<*}{\bot}\qquad (<snoc)\ \frac{r<\mathfrak{snoc}(s,\ t)}{\bot}$$

$$(<)_i\ \frac{}{t_i<\mathfrak{cons}(t_1,\ t_2)}\ i=1,\ 2\qquad (<cons)_i\ \frac{s<t_i}{s<\mathfrak{cons}(t_1,\ t_2)}\quad i=1,\ 2$$

$$(<cons\ E)\ \frac{r<\mathfrak{cons}(s,\ t)\qquad \overset{[r=s]}{c}\qquad \overset{[r<s]}{c}\qquad \overset{[r=t]}{c}\qquad \overset{[r<t]}{c}}{c}$$

As the final rule of inference for $\mathbb{BSA}$ we have the induction inference.

$$(ind)\ \frac{\mathscr{a}_z[\mathfrak{nil}]\qquad \overset{[\mathscr{a}_z[x]][\mathscr{a}_z[y]]}{\mathscr{a}_z[\mathfrak{cons}(x,\ y)]}\qquad \overset{[\mathscr{a}_z[x]][\mathscr{a}_z[y]]}{\mathscr{a}_z[\mathfrak{snoc}(x,\ y)]}}{\mathscr{a}_z[t]}$$

The assumptions discharged by this rule are called *induction hypotheses*. In this rule, the variables x and y must be distinct and must satisfy the *eigen variables condition*. Namely, the variables x and y may not occur free in $\mathscr{a}_z[\mathfrak{nil}]$ or in any assumption other than the induction hypotheses on which the premises $\mathscr{a}_z[\mathfrak{cons}(x,\ y)]$ and $\mathscr{a}_z[\mathfrak{snoc}(x,\ y)]$ depend.

3.6. interpretation

We now explain the intended interpretation of the theory $\mathcal{BSA}$. The intended domain of interpretation of our theory is $\mathbb{S}$. We first define the *denotation* $[\![t]\!]$ of a closed term t as follows.

1. $[\![\mathbf{nil}]\!] \equiv *$
2. $[\![\mathbf{cons}(s, t)]\!] \equiv [\, [\![s]\!] \,.\, [\![t]\!] \,]$
3. $[\![\mathbf{snoc}(s, t)]\!] \equiv (\, [\![s]\!] \,.\, [\![t]\!] \,)$

It should be clear that each closed term denotes a unique sexp, and for each sexp t there uniquely exists a closed term $\mathfrak{t}$ which denotes t.

We next assign a truth value (*true* or *false*) with each quantifier free closed wff. We first define the set of *descendants* of a sexp as follows.

1. The descendants of $*$ is empty.
2. The descendants of $[s\,.\,t]$ is the union of the descendants of s and t and the set $\{s, t\}$.
3. The descendants of $(s\,.\,t)$ is empty.

Thus, for instance, the descendants of $[[*]\,.\,(*\,.\,*)]$ is the set $\{*, [*], (*\,.\,*)\}$. We say that s is a *descendant* of t if s is a member of the descendants of t.

Let s and $\mathfrak{t}$ be closed terms and let s and t respectively be their denotations. Then the closed wff $s=\mathfrak{t}$ is *true* if s and t are the same sexp, and it is *false* if s and t are distinct. The closed wff $s<\mathfrak{t}$ is *true* if s is a descendant of t and is *false* otherwise.

Let $\mathfrak{a}$ be any closed quantifier free wff. Since it is a propositional combination of the atomic wffs of the above form, we can calculate its truth value by first replacing each atomic sub-wff by its value and then evaluating the resulting boolean expression in the usual way.

We now define the class of *primitive wffs* for which we can also assign truth values if they are closed.

1–2. If s and $\mathfrak{t}$ are terms then $s=\mathfrak{t}$ and $s<\mathfrak{t}$ are primitive wffs.

3–4. If $\mathfrak{a}_1,\ldots,\mathfrak{a}_n$ $(n\geqq 0)$ are primitive wffs then $\mathfrak{a}_1\wedge\cdots\wedge\mathfrak{a}_n$ and $\mathfrak{a}_1\vee\cdots\vee\mathfrak{a}_n$ are primitive wffs.

5. If $\mathfrak{a}_1,\ldots,\mathfrak{a}_n$ $(n\geqq 0)$ and $\mathfrak{b}$ are primitive wffs then $\mathfrak{a}_1,\ldots,\mathfrak{a}_n\rightarrow\mathfrak{b}$ is a primitive wff.

6–7. If $x_1,\ldots,x_n$ is a sequence of distinct pure variables, $\mathfrak{t}_1,\ldots,\mathfrak{t}_n$ is a sequence of terms, u_i $(1\leqq i\leqq n)$ is the elevation of $\mathfrak{t}_i$ with respect to $x_1,\ldots,x_n$ and $\mathfrak{a}$ is a primitive wff then $\forall(x_1,\ldots,x_n;\, x_1<u_1,$

$\ldots, x_n<u_n\to a)$ and $\exists(x_1,\ldots, x_n; x_1<u_1\wedge\cdots\wedge x_n<u_n\to a)$ are primitive wffs.

The primitive wffs defined by the clauses 6 and 7 above will respectively be abbreviated as:

$$\forall(x_1<t_1,\ldots, x_n<t_n; a)$$
$$\exists(x_1<t_1,\ldots, x_n<t_n; a)$$

(We will use this abbreviation for any wff a as well.) Since for each sexp t we can calculate the set of its descendants which is a finite set, it should be clear that we can uniquely assign a truth value for each closed primitive wff.

Next, we define *Σ-wffs* as follows:

1. A primitive wff is a Σ-wff.

2–3. If $a_1,\ldots, a_n$ $(n\geqq 0)$ are Σ-wffs then $a_1\wedge\cdots\wedge a_n$ and $a_1\vee\cdots\vee a_n$ are Σ-wffs.

4. If $a_1,\ldots, a_n$ $(n\geqq 0)$ are primitive wffs and b is a Σ-wff then $a_1,\ldots, a_n\to b$ is a Σ-wff.

5. If $x_1,\ldots, x_n$ is a sequence of distinct pure variables and a is a Σ-wff then $\exists(x_1,\ldots, x_n; a)$ is a Σ-wff.

We can define the *truth* of a closed Σ-wff inductively. The definition for the cases 1–4 is given similarly as for primitive wffs. For the case 5, we give the following definition. A closed Σ-wff $\exists(x_1,\ldots, x_n; a)$ is defined to be *true* if we can find a sequence of closed terms $t_1,\ldots, t_n$ for which $a_{x_1,\ldots,x_n}[\![t_1,\ldots, t_n]\!]$ becomes *true*.

We may say that $\mathcal{BSA}$ is *correct* if any closed Σ-wff which is provable in $\mathcal{BSA}$ is *true*. In this paper we assume the correctness of $\mathcal{BSA}$ without any further arguments. In particular we assume that $\mathcal{BSA}$ is *consistent* in the sense that there is no proof of the wff $\perp$.

3.7. BSA as a formal system

We now define BSA as a formal system and then define an isomorphism from $\mathcal{BSA}$ to BSA. It is possible to regard this isomorphism as an (symbolic) arithmetization of $\mathcal{BSA}$. Here we will not define the concept of proof in BSA since we give a full description of BSA as a formal axiom system in the next Section.

Let *Non_member*, *Pure_variable*, *Pure_variable_list*, *Variable*, *Term*, *Wff* and *Wff_list* respectively be the following sexps.

```
+ non_member
      | x, [ ]
      | x, [y . X]
            – ne[x, y]
            – non_member[x, X]
      ;
+ pure_variable
      | (: var . t)
      ;
+ pure_variable_list
      | [ ]
      | [x . X]
            – pure_variable[x]
            – non_member[x, X]
            – pure_variable_list[X]
      ;
+ variable
      | x
            – pure_variable[x]
      | (: free . x)
            – variable[x]
      ;
+ term
      | *
      | x
            – variable[x]
      | [s . t]
            – term[s]
            – term[t]
      | (: snoc . [s, t])
            – term[s]
            – term[t]
      ;
+ wff
      | eq[s, t]
            – term[s]
```

```
                    – term[t]
            | lt[s, t]
                    – term[s]
                    – term[t]
            | and[.A]
                    – wff_list[A]
            | or[.A]
                    – wff_list[A]
            | imply[A, b]
                    – wff_list[A]
                    – wff[b]
            | all[(: abs.[X, a])]
                    – pure_variable_list[X]
                    – wff[a]
            | ex[(: abs.[X, a])]
                    – pure_variable_list[X]
                    – wff[a]
            ;
    + wff_list
            | [ ]
            | [a.A]
                    – wff[a]
                    – wff_list[A]
            ;
```

Then the formal system:

$\mathbb{BSA}_0 \equiv$ [*Ne*, *Non_member*, *Pure_variable*, *Pure_variable_list*, *Variable*, *Term*, *Wff*, *Wff_list*]

defines basic concepts in $\mathbb{BSA}$. Thus, for instance, we say that (a sexp) a is a *wff* if $\vdash_{\mathbf{BSA}_0}$ wff[a] holds.

Example 3.2.

(: ∗ . ∗) is a term since we have $\vdash_{\mathbf{BSA}_0}$ term[(: ∗ . ∗)]. □

In this way we can continue to give a complete definition of $\mathbb{BSA}$ as a formal system. But as we said earlier we will not do so here because we will give a complete definition of $\mathbb{BSA}$ in the next Section.

We now explain that the concepts which we defined formally here are es-

sentially the same as the corresponding concepts which we defined for $\mathcal{BSA}$. To this end we define a translation from syntactic objects like terms or wffs in $\mathcal{BSA}$ into $\mathbb{S}$. We denote the translation of $\mathcal{A}$ by $\mathcal{A}^\dagger$.

Terms in $\mathcal{BSA}$ are translated as follows.

1.1. $\mathbf{var}_t^\dagger$ is $(\mathrm{var}\,.\,t)$.

1.2. $\mathbf{free}(x)^\dagger$ is $(\mathrm{free}\,.\,x^\dagger)$.

2. $\mathbf{nil}^\dagger$ is $*$.

3–4. $\mathbf{snoc}(s, t)^\dagger$ is $[s^\dagger, t^\dagger]$ and $\mathbf{snoc}(s, t)^\dagger$ is $(: s^\dagger, t^\dagger)$.

The translation of wffs in $\mathcal{BSA}$ is defined as follows.

1–2. $\mathbf{eq}(s, t)^\dagger$ is $\mathrm{eq}[s^\dagger, t^\dagger]$ and $\mathbf{lt}(s, t)^\dagger$ is $\mathrm{lt}[s^\dagger, t^\dagger]$.

3–4. $\mathbf{and}(\mathcal{A}_1, \ldots\ \mathcal{A}_n)^\dagger$ is $\mathrm{and}[\mathcal{A}_1^\dagger, \ldots\ \mathcal{A}_n^\dagger]$ and $\mathbf{or}(\mathcal{A}_1, \ldots, \mathcal{A}_n)^\dagger$ is $\mathrm{or}[\mathcal{A}_1^\dagger, \ldots, \mathcal{A}_n^\dagger]$.

5. $\mathbf{imply}((\mathcal{A}_1, \ldots, \mathcal{A}_n), \mathcal{B})^\dagger$ is $\mathrm{imply}[[\mathcal{A}_1^\dagger, \ldots, \mathcal{A}_n^\dagger], \mathcal{B}^\dagger]$.

6–7. $\mathbf{all}((x_1, \ldots, x_n), \mathcal{A})^\dagger$ is $\mathrm{all}[\mathrm{abs}\,.\,[[x_1^\dagger, \ldots, x_n^\dagger], \mathcal{A}^\dagger])]$ and $\mathbf{exist}((x_1, \ldots, x_n), \mathcal{A})^\dagger$ is $\mathrm{ex}[(\mathrm{abs}\,.\,[[x_1^\dagger, \ldots, x_n^\dagger], \mathcal{A}^\dagger])]$.

It is then easy to verify that this translation sends each syntactic entity in $\mathcal{BSA}$ into corresponding entity in $\mathbb{BSA}$. Thus if $\mathcal{A}$ is a wff in the sense of $\mathcal{BSA}$ then $\mathcal{A}^\dagger$ is a wff in $\mathbb{BSA}$, that is, we have $\vdash_{\mathbb{BSA}_0} \mathrm{wff}[\mathcal{A}^\dagger]$. Moreover for each wff a in $\mathbb{BSA}$ we can uniquely find a wff $\mathcal{A}$ in $\mathcal{BSA}$ such that $\mathcal{A}^\dagger$ is a. A similar correspondence holds also for terms. It is also obvious from our definition that the translation is homomorphic with respect to the inductive definition of syntactic entities. We may thus conclude that both $\mathcal{BSA}$ and $\mathbb{BSA}$ give definitions to the abstract concepts such as terms or wffs in terms of their respective representations. For this reason we will use the same abbreviations which we used for syntactic entities in $\mathcal{BSA}$ as abbreviations for the corresponding objects in $\mathbb{BSA}$. We will also use syntactic variables to make our intention clear. Thus for instance if in some context we wish to refer a certain sexp as a wff, we will use syntactic variables $\mathcal{A}$, $\mathcal{B}$ or $\mathcal{C}$ for it.

Example 3.3.

$\forall(x; \exists(x; x = \sharp x))$ is an abbreviation of the sexp

$$\mathrm{all}[(\mathrm{abs}\,.\,[[x], \mathrm{ex}[(\mathrm{abs}\,.\,[[x], \mathrm{eq}[x, (\mathrm{free}\,.\,x)]])]])]$$

which is a wff in $\mathbb{BSA}$. □

§4. First Order Theories

In this section we introduce the formal system FOT (for *F*irst *O*rder *T*heory). In FOT one can define a general class of first order theories including BSA and its extensions or restrictions. The basic idea is that any sexp S when viewed as a formal system can be used to define a formal theory in countable first-order language. We first introduce the concept of quasi-quotation which provides a convenient notation for various syntactic entities. We then define FOT using this notation.

4.1. quasi-quotation

When one makes a statement about an object one must use a name for that object to refer to that object. If that object is linguistic it is possible to use that object directly as the name of that object and such usage of linguistic object is called *autonymous*. Autonymous usage of an object as the name for it is harmless provided the symbols appearing in that linguistic object are not included in the vocabulary of the meta language, that is the language in which the statement is made. Another method of obtaining a name for a linguistic object which is applicable to most cases, is to *quote* that linguistic object. (Still, there is a problem of how to quote quotation marks. This problem is solved by computer scientists in many ways.)

The mechanism of quotation is so restrictive, and it is often desirable to be able to unquote part of the quotation. Quasi-quotation is such a mechanism and Quine used it in his book [9] extensibly. Let us give an example of quasi-quotation. The fourth clause of our inductive definition of terms in 3.3 is as follows.

4. If s and t are terms then $\mathrm{smoc}(s, t)$ is a term.

Using corners which are Quine's notational device for quasi-quotation, the above clause will look like this:

4. If s and t are terms then $\ulcorner \mathrm{smoc}(s, t) \urcorner$ is a term.

The difference of quasi-quotation from ordinary quotation is this. Within quasi-quotation marks (corners in this case) one can embed meta variables ('s' and 't' in this case) among the symbols of the object language. Such a quasi-quotation designates an expression that result from the content of the quasi-quotation by replacing each meta variable by what it designates. Thus if both

s and t designates '𝔫𝔦𝔩' then $\ulcorner$𝔠𝔬𝔫𝔰(s, t)$\urcorner$ designates the expression '𝔠𝔬𝔫𝔰(𝔫𝔦𝔩, 𝔫𝔦𝔩)'. This is the basic idea of quasi-quotation. However, with a slight danger of falling into the trap of *use* and *mention* confusion (Quine [9], pp. 23–26.), we have avoided the use of quasi-quotation nor quotation for the ease of readability.

Quasi-quotation is a familiar idea among users of computers who use editors, text-formatters etc. (See, e.g., Bourne [2].) Some LISP languages like Maclisp have a mechanism of quasi-quotation as an input macro known as *back quote macro*. For example, if the variables S and T both has NIL as its current value then evaluating the expression:

```
`(, S . , T)
```

will return the value (NIL . NIL). Since in LISP the same symbols are used both in the meta language and the object language, a comma is placed within the scope of back quote to tell the input routine that the following expression should be evaluated (rather than quoted).

We will introduce Maclisp like quasi-quotation mechanism as a convenient notational device for referring to various syntactic objects within a formal system. In connection with this, we note that we have incorporated the mechanism of quotation as a part of evaluation mechanism. (Recall (E6) of the definition of *eval*.) We define our quasi-quotation mechanism formally by the formal system [*Quasi_quote*, *Ne*], where *Quasi_quote* is the following sexp and *Ne* was defined in 𝕌𝕟𝕚𝕧:

```
+ quasi_quote
    | *, *
    | [s . t], [u . v]
          – quasi_quote[s, u]
          – quasi_quote[t, v]
    | (: s . t), (: snoc . [u, v])
          – ne[s, *]
          – ne[s, eval]
          – quasi_quote[s, u]
          – quasi_quote[t, v]
    | (: * . t), (: * . t)
    | (: eval . t), t
    ;
```

We can easily verify that for any sexp t there uniquely exists a sexp v such that:

$$\vdash_{[Quasi\_quote,Ne]} \text{quasi\_quote}[t, v]$$

In this case we call v the *quasi-quote expansion* of t. We introduce the following abbreviations.

$`t$ for the quasi-quote expansion of t

$/t$ for (eval . t)

Example 4.1.

$$\begin{aligned}`(: /s . /t) &\equiv `(\text{snoc} . [/s, /t]) \\ &\equiv (\text{snoc} . (`\text{snoc} . `[/s. /t])) \\ &\equiv (\text{snoc} . (\text{snoc} . [`/s, `/t])) \\ &\equiv (\text{snoc} . (\text{snoc} . [s, t])) \\ &\equiv (: \text{snoc} . [s, t]) \quad \square\end{aligned}$$

We can therefore rewrite the sexp *Term* in the formal system $\mathbb{BSA}_0$ as follows.

```
+ term
      | *
      | x
            – variable[x]
      | [s.t]
            – term[s]
            – term[t]
      | `(: /s . /t)
            – term[s]
            – term[t]
      ;
```

The reader should compare this with the definition of terms in 3.3 and consider the reason why quasi-quotation is not necessary for *cons* in this formal definition of terms. □

Note. We have introduced the mechanism of quasi-quotation as a part of evaluation mechanism in our former programming languages for the domain $\mathbb{S}_{old}$. (See [10], [11].) Edinburgh LCF [6] also uses quasi-quotation as convenient notation for its PPλ objects. □

4.2. FOT

We define the formal system FOT as a list of 39 sexps of which first 7 are taken from the formal system Univ and the remaining sexps are the following 32 sexps.

```
+ member
    | x, [x.X]
    | x, [y.X]
        - member[x, X]
    ;
+ non_member
    | x, [ ]
    | x, [y.X]
        - ne[x, y]
        - non_member[x, X]
    ;
+ pure_variable
    | (: var.t)
    ;
+ pure_variable_list
    | [ ]
    | [x.X]
        - pure_variable[x]
        - non_member[x, X]
        - pure_variable_list[X]
    ;
+ variable
    | x
        - pure_variable[x]
    | (: free.x)
        - variable[x]
    ;
+ pure_part
    | (: var.t), (: var.t)
    | (: free.x), y
```

```
            – pure_part[x, y]
      ;
+ length
      | *, *
      | [x . X], [* . n]
            – length[X, n]
      ;
+ symbol
      | (: * . t)
      ;
+ term
      | x, S
            – variable[x]
      |*, S
            – theorem[special[nil], S]
      | [s . t], S
            – theorem[special[cons], S]
            – term[s, S]
            – term[t, S]
      | '(: /s . /t), S
            – theorem[special[snoc], S]
            – term[s, S]
            – term[t, S]
      | (: apply . [Fun . T]), S
            – symbol[Fun]
            – term_list[T, S]
            – length[T, Arity]
            – theorem[function[[Fun . Arity]], S]
      | (: quote . t), S
            – theorem[special[quote], S]
      | (: * . t), S
            – theorem[special[symbol], S]
      ;
+ term_list
      | [ ], S
      | [t . T], S
```

```
                - term[t, S]
                - term_list[T, S]
        ;
+ admissible
        | p
                - symbol[p]
                - ne[p, and]
                - ne[p, or]
                - ne[p, imply]
                - ne[p, all]
                - ne[p, ex]
        ;
+ wff
        | eq[s, t], S
                - term[s, S]
                - term[t, S]
        | [Prd . T], S
                - admissible[Prd]
                - term_list[T, S]
                - length[T, Arity]
                - theorem[predicate[[Prd . Arity]], S]
        | and[ . A], S
                - wff_list[A, S]
        | or[ . A], S
                - wff_list[A, S]
        | imply[A, b], S
                - wff_list[A, S]
                - wff[b, S]
        | all[(: abs . [X, a])], S
                - pure_variable_list[X]
                - wff[a, S]
        | ex[(: abs . [X, a])], S
                - pure_variable_list[X]
                - wff[a, S]
        ;
+ wff_list
```

```
        | [ ], S
        | [a . A], S
                – wff[a, S]
                – wff_list[A, S]
        ;
+ find
        | [ ], [ ], x, x
        | [t . T], [x . X], x, t
        | [t . T], [y . X], x, v
                – ne[x, y]
                – find[T, X, x, v]
        ;
+ rename
        | x, y, x
                – pure_variable[x]
        | (: free . x), y, x
                – pure_variable[y]
        | (: free . x), (: free . y), (: free . z)
                – rename[x, y, z]
        ;
+ free
        | *, X, *
        | y, X, (: free . y)
                – variable[y]
                – pure_part[y, x]
                – member[x, X]
          | y, X, y
                – variable[y]
                – pure_part[y, x]
                – non_member[x, X]
        | [s . t], X, [u . v]
                – free[s, X, u]
                – free[t, X, v]
        | '(: /s . /t), X, '(: /u . /v)
                – free[s, X, u]
                – free[t, X, v]
```

```
        | (: *.t), X, (: *.t)
        | (: quote.t), X, (: quote.t)
        | (: apply.[f.T]), X, (: apply.[f.V])
                - free[T, X, V]
        | (: eval.t), X, (: eval.t)
        ;
+ subst
        | T, X, x, v
                - variable[x]
                - find[T, X, x, v]
        | T, X, *, *
        | T, X, [s.t], [u.v]
                - subst[T, X, s, u]
                - subst[T, X, t, v]
        | T, X, '(: /s./t), '(: /u./v)
                - subst[T, X, s, u]
                - subst[T, X, t, v]
        | T, X, (: *.t), (: *.t)
        | T, X, (: quote.t), (: quote.t)
        | T, X, (: apply.[f.U]), (: apply.[f.V])
                - subst[T, X, U, V]
        | T, X, (: abs.[Y, t]), (: abs.[Y, v])
                - pure_variable_list[Y]
                - free[T, Y, V]
                - free[X, Y, Z]
                - subst[V, Z, t, v]
        ;
+ subst 1
        | t, x, s, v
                - subst[[t], [x], s, v]
        ;
+ bind_find
        | [ ], [ ], x, x
        | [t.T], [x.X], x, t
        | [t.T], [y.X], x, v
```

```
            – ne[x, y]
            – pure_part[x, u]
            – pure_part[y, u]
            – rename[x, y, v]
      | [t.T], [y.X], x, v
            – ne[x, y]
            – pure_part[x, u1]
            – pure_part[y, u2]
            – ne[u1, u2]
            – bind_find[T, X, x, v]
      ;
+ bind_subst
      | T, X, x, v
            – variable[x]
            – bind_find[T, X, x, v]
      | T, X, *, *
      | T, X, [s.t], [u.v]
            – bind_subst[T, X, s, u]
            – bind_subst[T, X, t, v]
      | T, X, '(: /s./t), '(: /u, /v)
            – bind_subst[T, X, s, u]
            – bind_subst[T, X, t, v]
      | T, X, t, t
            – symbol[t]
      | T, X, (: quote.t), (: quote.t)
      | T, X, (: apply.[f.U]), (: apply [f.V])
            – bind_subst[T, X, U, V]
      | T, X, (: abs.[Y, t]), (: abs.[Y, v])
            – pure_variable_list[Y]
            – free [T, Y, V]
            – free[X, Y, Z]
            – bind_subst[V, Z, t, v]
      ;
+ append
      | [ ], Y, Y
      | [x.X], Y, [x.Z]
```

```
                - append[X, Y, Z]
        ;
+ addend
        | [ ], Y, Y
        | [x.X], Y, Z
                - addend[X, Y, Z]
                - member[x, Z]
        | [x.X], Y, [x.Z]
                - addend[X, Y, Z]
                - non_member[x, Z]
        ;
+ vars
        | x, [x]
                - variable[x]
        | *, [ ]
        | [s.t], V
                - vars[s, S]
                - vars[t, T]
                - addend[S, T, V]
        | '(: /s./t), V
                - vars[s, S]
                - vars[t, T]
                - addend[S, T, V]
        | (: *.t), [ ]
        | (: quote.t), [ ]
        | (: apply.[f.T]), V
                - vars[T, V]
        | (: abs.[X, a]), V
                - vars[a, U]
                - down[X, U, V]
        ;
+ down
        | X, [ ], [ ]
        | X, [x.U], V
                - pure_variable[x]
                - member[x, X]
```

```
            – down[X, U, V]
      | X, [x . U], [x . V]
            – pure_variable[x]
            – non_member[x, X]
            – down[X, U, V]
      | X, [(: free . y) . U], [y . V]
            – pure_part[y, z]
            – member[z, X]
            – down[X, U, V]
      | X, [(: free . y) . U], [(: free . y) . V]
            – pure_part[y, z]
            – non_member[z, X]
            – down[X, U, V]
      ;
+ new
      | x, t
            – vars[t, X]
            – non_member[x, X]
      ;
+ new_list
      | *, t
      | [x . X], t
            – new[x, t]
            – new_list[X, t]
      ;
+ eq_pr
      | S, E, [ ], [ ], [ ]
      | S, E, [p . P], [u . U], [v . V]
            – pr[S, E, p, u=v]
            – eq_pr[S, E, P, U, V]
      ;
+ lpr
      | S, E, [ ], [ ]
      | S, E, [p . P], [a . A]
            – pr[S, E, p, a]
            – lpr[S, E, P, A]
```

```
        ;
    + upr
        | S, [ ], E, [ ], c
        | S, [a . A], E, [p . P], c
                - pr[S, [a . E], p, c]
                - upr[S, A, E, P, c]
        ;
    + axiom
        | a, S
                - proves[p, axiom[a], S]
                - wff[a, S]
        ;
    + pr
        | S, E, [a, axiom[ ]], a
                - axiom[a, S]
                - wff_list[E, S]
        | S, E, [a, assumption[ ]], a
                - member[a, E]
                - wff_list[E, S]
        | S, E, [and[ . A], and_I[ . P]], and [ . A]
                - lpr[S, E, P, A]
        | S, E, [a, and_E[p]], a
                - pr[S, E, p, and[ . A]]
                - member[a, A]
        | S, E, [or[ . A], or_I[p]], or[ . A]
                - pr[S, E, p, a]
                - member[a, A]
                - wff_list[A, S]
        | S, E, [a, or_E[p . P]], a
                - pr[S, E, p, or[ . A]]
                - upr[S, A, E, P, a]
        | S, E, [imply[A, b], imply_I[p]], imply[A, b]
                - append[A, E, F]
                - pr[S, F, p, b]
        | S, E, [a, imply_E[p, P]], a
                - pr[S, E, p, imply[A, a]]
```

```
            – lpr[S, E, P, A]
        | S, E, [all[(: abs . [X, a])], all_I[[X, Y], p]], all(: abs . [X, a])]
            – new_list[Y, E]
            – new_list[Y, (: abs . [X, a])]
            – pure_variable_list[X]
            – wff[a, S]
            – bind_subst[Y, X, a, b]
            – pr[S, E, p, b]
        | S, E, [a, all_E[T, p]], a
            – pr[S, E, p, all[(: abs . [X, b])]]
            – term_list[T, S]
            – bind_subst[T, X, b, a]
        | S, E, [ex[(: abs . [X, b])], ex_I[T, p]], ex[(: abs . [X, b])]
            – term_list[T, S]
            – pure_variable_list[X]
            – wff[b, S]
            – bind_subst[T, X, b, a]
            – pr[S, E, p, a]
        | S, E, [a, ex_E[Y, p, q]], a
            – new_list[Y, E]
            – new_list[Y, a]
            – pr[S, E, p, ex[(: abs . [X, b])]]
            – bind_subst[Y, X, b, c]
            – pr[S, [c . E], q, a]
        | S, E, [t=t, axiom_id[ ]], t=t
            – term[t, S]
        | S, E, [a, subst[[U, V, X, b], p . P]], a
            – wff[b, S]
            – subst[U, X, b, c]
            – subst[V, X, b, a]
            – pr[S, E, p, c]
            – eq_pr[S, E, P, U, V]
        ;
+ thm
        | a, S
            – pr[S, [ ], p, a]
```

;

Let us briefly explain the intended meanings of some important concepts defined in FOT. We will call any formal system an *axiom system*. Then term[t, S] means that t is a *term* in the axiom system S. Similarly, wff[a, S] means that a is a *wff* in S. Note that here S is used to specify the language. The operation of substitution is defined by the predicate 'subst'. Namely, subst[T, X, t, v] menas that v is the result of simultaneously substituting members of the list T for the free occurrences of the corresponding members of the list X in t. The uniqueness of the result of substitution is guaranteed by the following easily verifiable fact:

$$\vdash_{\mathrm{FOT}}\mathrm{subst}[T, X, t, v_1],\ \vdash_{\mathrm{FOT}}\mathrm{subst}[T, X, t, v_2] \Longrightarrow v_1 \equiv v_2$$

Let $T\equiv[t_1,\ldots, t_n]$ and $X\equiv[x_1,\ldots, x_n]$. If $\vdash_{\mathrm{FOT}}\mathrm{subst}[T, X, t, v]$ holds for some v then such a v is unique by the above fact. We will denote this v by:

$$t_{x_1,\ldots,x_n}[t_1,\ldots, t_n]$$

The meaning of vars[t, X] is that X is the list of variables occurring free in t. We will say that x is *new* to t if new[x, t] holds where new[x, t] means that x does not occur free in t. Finally, pr[S, E, p, a] means that p is a *proof* in the axiom system S of a from the assumption E.

It should be clear that arbitrary formal theory with countable first-order language over intuitionistic logic (or classical logic) can be treated in this framework. As the first concrete example of an axiom system we now define the axiom system BSA as follows:

BSA≡[*Special*, *Predicate*, *Axiom*, *Bas_term*, *Bsa_wff*, *Bsa_wff_list*, *Variable*, *Pure_variable*, *Pure_variable_list*, *Non_member*, *Ne*, *New*, *Vars*, *Addend*, *Member*, *Subst*, *Find*, *Free*, *Pure_part*, *Subst1*]

where we have defined the last 14 sexps already and the first 6 sexps are respectively as follows.

```
+ special
      | nil
      | cons
      | snoc
      ;
+ predicate
```

```
      | lt[*, *]
      ;
+ axiom
      | '∀(s, t; [s.t] ≠ *)
      | '∀(s, t; (: s.t) ≠ *)
      | '∀(s, t, u, v; [s.t] ≠ (: u.v))
      | '∀(s, t, u, v; [s.t] = [u.v] → s = u)
      | '∀(s, t, u, v; [s.t] = [u.v] → t = v)
      | '∀(s, t, u, v; (: s.t) = (: u.v) → s = u)
      | '∀(s, t, u, v; (: s.t) = (: u.v) → t = v)
      | '∀(r; ¬(r < *))
      | '∀(r, s, t; ¬(r < (: s.t)))
      | '∀(s, t; s < [s.t])
      | '∀(s, t; t < [s.t])
      | '∀(r, s, t; r < s → r < [s.t])
      | '∀(r, s, t; r < t → r < [s, t])
      | '∀(r, s, t; r < [s.t] → r = s ∨ r < s ∨ r = t ∨ r < t)
      | '(/a0, ∀(/x, /y; /IHx, /IHy → /a1), ∀(/x, /y; /IHx, /IHy→/a2)
      → /b)
            – pure_variable[x]
            – pure_variable[y]
            – pure_variable[z]
            – ne[x, y]
            – bsa_wff[a]
            – new[x, a]
            – new[y, a]
            – bas_term[t]
            – substl[t, z, a, b]
            – substl[*, z, a, a0]
            – substl[x, z, a, IHx]
            – substl[y, z, a, IHy]
            – substl[[x.y], z, a, a1]
            – substl['(: /x./y), z, a, a2]
+ bsa_term
      | x
            – variable[x]
```

```
        | *
        | [s.t]
              - bsa_term[s]
              - bsa_term[t]
        | `(: /s./t)
              - bsa_term[s]
              - bsa_term[t]
        ;
  + bsa_wff
        | eq[s, t]
              - bsa_term[s]
              - bsa_term[t]
        | lt[s, t]
              - bsa_term[s]
              - bsa_term[t]
        | and[.A]
              - bsa_wff_list[A]
        | or[.A]
              - bsa_wff_list[A]
        | imply[A, b]
              - bsa_wff_list[A]
              - bsa_wff[b]
        | all[(: abs.[X, a])]
              - pure_variable_list[X]
              - bsa_wff[a]
        | ex[(: abs.[X, a])]
              - pure_variable_list[X]
              - bsa_wff[a]
        ;
  + bsa_wff_list
        | [ ]
        | [a.A]
              - bsa_wff[a]
              - bsa_wff_list[A]
        ;
```

We now introduce notations for some concepts which we defined in FOT.

Let S be an axiom system. We will say that p is a *proof* of a from the *assumptions* E in S if

$$\vdash_{\mathrm{FOT}}\mathrm{pr}[S, E, p, a]$$

holds, and we use the notation

$$S\colon p\vdash a \text{ for } \vdash_{\mathrm{FOT}}\mathrm{pr}[S, [\ ], p, a]$$

Similarly we will say that a is a *theorem* of S if

$$S\colon p\vdash a$$

holds for some p and use the notation

$$S\vdash a$$

for it.

We are now using the provability sign '$\vdash$' for two purposes. Namely, one usage is for the provability in formal systems and the other is for the provability in axiom systems. However, these usages can always be distinguished syntactically by the presence or absence of a subscript for the provability sign.

Let S and T be axiom systems. We say that T is an *extension* of S if for any sexp a we have:

(i) if $\vdash_{\mathrm{FOT}}\mathrm{wff}[a, S]$ then $\vdash_{\mathrm{FOT}}\mathrm{wff}[a, T]$ and

(ii) if $S\vdash a$ then $T\vdash a$.

(The condition (i) is, in fact, redundant since it follows from (ii).) T is said to be a *conservative extension* of S if:

(i) T is an extension of S and

(ii) if $\vdash_{\mathrm{FOT}}\mathrm{wff}[a, S]$ and $T\vdash a$ then $S\vdash a$.

S is said to be *consistent* if for no p

$$S\colon p\vdash \bot$$

holds. It is clear that if S is consistent and T is a conservative extension of S then T is also consistent.

§5. The Axiom System SA

In this Section we introduce an axiom system SA which is a conservative extension of BSA. The motivation for the extension of the system is to obtain a system which is powerful enough to let one work actually within the system

comfortably. Our final goal is to get a system in which one can formally carry out all the mathematical and metamathematical arguments we are informally doing in this paper. The introduction of SA is a first step toward this goal.

5.1. SA

The axiom system SA is defined as the sexp:

SA ≡ [*Special*, *Predicate*, *Axiom*, *Sa_term*, *Sa_wff*, *Sa_wff_list*, *Variable*, *Pure_variable*, *Pure_variable_list*, *Non_member*, *Ne*, *New*, *Vars*, *Addend*, *Member*, *Subst*, *Find*, *Free*, *Prue_part*, *Substl1*]

where the last 14 members of SA are already defined elsewhere and the first 6 are listed below. The number enclosed between '%' symbols are comments, and we will use this number as the number of the axiom defined by the corresponding clause.

```
+ special
      | nil
      | cons
      | snoc
      | quote
      | symbol
      ;
+ predicate
      | lt[*, *]
      ;
+ axiom
      | 'V(s, t; [s.t] ≠ *) % 1 %
      | 'V(s, t; (: s.t) ≠ *) % 2 %
      | 'V(s, t, u, v; [s.t] ≠ (: u.v)) % 3 %
      | 'V(s, t, u, v; [s.t] = [u.v] → s = u) % 4 %
      | 'V(s, t, u, v; [s.t] = [u.v] → t = v) % 5 %
      | 'V(s, t, u, v; (: s.t) = (: u.v) → s = u) % 6 %
      | 'V(s, t, u, v; (: s.t) = (: u.v) → t = v) % 7 %
      | 'V(r; ¬(r < *)) % 8 %
      | 'V(r, s, t; ¬(r < (: s.t))) % 9 %
      | 'V(s, t; s < [s.t]) % 10 %
```

```
| '∀(s, t; t < [s.t]) % 11 %
| '∀(r, s, t; r < s → r < [s.t]) % 12 %
| '∀(r, s, t; r < t → r < [s.t]) % 13 %
| '∀(r, s, t; r < [s.t] → r = s ∨ r < s ∨ r = t ∨ r < t) % 14 %
| '((quote.*) = *) % 15 %
| `((quote.[/s./t]) = [(quote./s).(quote./t)]) % 16 %
| `((quote.(/s./t)) = (:(quote./s).(quote./t))) % 17 %
| '((*.*) = (:*.*)) % 18 %
| `((*./s) = (:*./u), (*./t) = (:*./v) →
            (*.[/s./t]) = (:*.[/u./v])) % 19 %
| `((*./s) = (:*./u), (*./t) = (:*./v) →
            (*.(/s./t)) = (:*.(:/u./v))) % 20 %
| `(/a0, ∀(/x, /y; /IHx, /IHy → /a1),
      ∀(/x, /y; /IHx, /IHy → /a2) → /b) % 21 %
      – pure_variable[x]
      – pure_variable[y]
      – pure_variable[z]
      – ne[x, y]
      – sa_wff[a]
      – new[x, a]
      – new[y, a]
      – sa_term[t]
      – substl[t, z, a, b]
      – substl[*, z, a, a0]
      – substl[x, z, a, IHx]
      – substl[y, z, a, IHy]
      – substl[[x.y], z, a, a1]
      – substl[`(:/x./y), z, a, a2]
| `(∀(/z; ∀(/w; /w < /z → /a1) → /a) → /b) % 22 %
      – pure_varaiable[z]
      – pure_variable[w]
      – ne[z, w]
      – sa_wff[a]
      – new[w, a]
      – sa_term[t]
      – substl[w, z, a, a1]
```

```
                — substl[t, z, a, b]
        ;
+ sa_term
        | x
                — variable[x]
        | *
        | [s.t]
                — sa_term[s]
                — sa_term[t]
        | '(: /s./t)
                — sa_term[s]
                — sa_term[t]
        | (: *.t)
        | (: quote.t)
        ;
+ sa_wff
        | eq[s, t]
                — sa_term[s]
                — sa_term[t]
        | lt[s, t]
                — sa_term[s]
                — sa_term[t]
        | and[.A]
                — sa_wff_list[A]
        | or[.A]
                — sa_wff_list[A]
        | imply[A, b]
                — sa_wff_list[A]
                — sa_wff[b]
        | all[(: abs.[X, a])]
                — pure_variable_list[X]
                — sa_wff[a]
        | ex[(: abs.[X, a])]
                — pure_variable_list[X]
                — sa_wff[a]
        ;
```

```
+ sa_wff_list
      | [ ]
      | [a . A]
            – sa_wff[a]
            – sa_wff_list[A]
      ;
```

Axiom 22 is the induction schema with respect to the ordering $<$ and we will refer to it as the *$<$-induction*

5.2. SA is conservative over BSA

In this subsection we show that SA is conservative over BSA. To prove this we consider an intermediate axiom system SA_0 which results from SA by simply deleting the $<$-induction schema (that is, axiom 22). We first show that SA_0 is conservative over BSA and then prove that SA is conservative over SA_0.

We prove the first part in the following form. We first define a mapping $(-)^*$ which sends a wff in SA_0 to a wff in BSA and then prove that this mapping has the following properties which are sufficient to prove that SA_0 is conservative over BSA.

(i) If SA_0: $\mathfrak{p} \vdash \mathfrak{a}$ then BSA: $\mathfrak{q} \vdash \mathfrak{a}^*$ for some $\mathfrak{q}$.

(ii) If $\mathfrak{a}$ is a wff in BSA then $\mathfrak{a}^*$ is $\mathfrak{a}$.

(We will use $\mathfrak{p}$ and $\mathfrak{q}$ as syntactic variables for proofs.)

We first translate terms in SA_0 to terms in BSA. The translation sends variables to the same variables, $*$ to $*$ and is homomorphic with respect to *cons* and *snoc*. The rest of the translation is defined as follows.

5.1. $(\mathrm{quote}\,.\,*)^* \equiv *$

5.2. $(\mathrm{quote}\,.\,[s\,.\,t])^* \equiv [(\mathrm{quote}\,.\,s)^*\,.\,(\mathrm{quote}\,.\,t)^*]$

5.3. $(\mathrm{quote}\,.\,(s\,.\,t))^* \equiv (:\,(\mathrm{quote}\,.\,s)^*\,.\,(\mathrm{quote}\,.\,t)^*)$

6.1. $(*\,.\,*)^* \equiv (:\,*\,.\,*)$

6.2. If $(*\,.\,s)^* \equiv (:\,*\,.\,u)$ and $(*\,.\,t)^* \equiv (:\,*\,.\,v)$ then $(*\,.\,[s\,.\,t])^* \equiv (:\,*\,.\,[u\,.\,v])$.

6.3. If $(*\,.\,s)^* \equiv (:\,*\,.\,u)$ and $(*\,.\,t)^* \equiv (:\,*\,.\,v)$ then $(*\,.\,(s\,.\,t))^* \equiv (:\,*\,.\,(:\,u\,.\,v))$.

We then extend $(-)^*$ homomorphically to wffs. It is easy to see that if $\mathfrak{a}$ is a wff in SA_0 then $\mathfrak{a}^*$ is a wff in BSA and that this mapping has the property (ii) above. We now prove (i) by the (informal) $<$-induction on $\mathfrak{p}$. More precisely,

we prove the following proposition by the $<$-induction on p.

$$\vdash_{\mathbf{FOT}} \mathrm{pr}[\mathbb{SA}_0, E, p, a]$$
$$\Longrightarrow \vdash_{\mathbf{FOT}} \mathrm{pr}[\mathbb{BSA}, E^*, q, a^*] \text{ for some } q \text{ and}$$
$$\vdash_{\mathbf{FOT}} \mathrm{lpr}[\mathbb{SA}_0, E, p, a]$$
$$\Longrightarrow \vdash_{\mathbf{FOT}} \mathrm{lpr}[\mathbb{BSA}, E^*, q, a^*] \text{ for some } q \text{ and}$$
$$\vdash_{\mathbf{FOT}} \mathrm{upr}[\mathbb{SA}_0, A, E, p, a]$$
$$\Longrightarrow \vdash_{\mathbf{FOT}} \mathrm{upr}[\mathbb{BSA}, A^*, E^*, q, a^*] \text{ for some } q \text{ and}$$
$$\vdash_{\mathbf{FOT}} \mathrm{eq\_pr}[\mathbb{SA}_0, E, p, U, V]$$
$$\Longrightarrow \vdash_{\mathbf{FOT}} \mathrm{eq\_pr}[\mathbb{BSA}, E^*, q, U^*, V^*] \text{ for some } q.$$

In proving this proposition we may assume the proposition for any sexp which is a descendant of p. Since this proposition is a conjunction of four propositions we must prove each of them. To prove the first proposition let us assume:

$$\vdash_{\mathbf{FOT}} \mathrm{pr}[\mathbb{SA}_0, E, p, a]$$

Then according to the definition of pr in $\mathbb{FOT}$ we have 14 cases to consider.

(Case 1) In this case we have for some p_0:

$$p \equiv [a, \mathrm{axiom}[p_0]] \tag{1.1}$$

$$\vdash_{\mathbf{FOT}} \mathrm{proves}[p_0, \mathrm{axiom}[a], \mathbb{SA}_0] \tag{1.2}$$

$$\vdash_{\mathbf{FOT}} \mathrm{wff}[a, \mathbb{SA}_0] \tag{1.3}$$

and

$$\vdash_{\mathbf{FOT}} \mathrm{wff\_list}[E, \mathbb{SA}_0] \tag{1.4}$$

Since Theorem 2.1 still holds even if we substitute $\mathbb{FOT}$ for $\mathbb{Univ}$ in it, we have by (1.3):

$$p_0 \vdash_{\mathbf{SA}_0} \mathrm{axiom}[a] \tag{1.5}$$

We will prove that we can either find a q_0 for which we have

$$q_0 \vdash_{\mathbf{BSA}} \mathrm{axiom}[a^*] \tag{1.6}$$

or find a q_0 for which we have

$$\vdash_{\mathbf{FOT}} \mathrm{pr}[\mathbb{BSA}, E^*, q_0, a^*] \tag{1.7}$$

According to the definition of axiom in $\mathbb{SA}_0$ we have 21 cases to consider. For the first 14 cases, a^* is identical to a and we may take q_0 as p_0 to get (1.6). We next consider the remaining cases.

(Case 1.15) In this case a is $((\mathrm{quote}.*) = *)$ and a^* is $(* = *)$. Putting q_0 as

$[* = *$, axiom_id[ ]] we get (1.7).

(Case 1.16) In this case a is ((quote . $[s . t]$) = [(quote . s) . (quote . t)]) for some s and t and a^* is $(T = T)$ where T is [(quote . s) . (quote . t)]. Putting q_0 as $[T = T$, axiom_id[ ]] we get (1.7).

(Case 1.17) is similar to (Case 1.16) and we omit it.

(Case 1.18) is similar to (Case 1.15) and we omit it.

(Case 1.19) In this case a is:

$$(* . s) = (: * . u), (* . t) = (: * . v) \to (* . [s . t]) = (: * . [u . v])$$

and a^* is:

$$(* . s)^* = (: * . u), (* . t)^* = (: * . v) \to (* . [s . t])^* = (: * . [u . v])$$

By the definition of $(-)^*$ we can find S and T such that:

$$(* . s)^* \equiv (: * . S) \text{ and } (* . t)^* \equiv (: * . T)$$

We can now rewrite a^* as:

$$(: * . S) = (: * . u), (: * . T) = (: * . v) \to (: * . [S . T]) = (: * . [u . v])$$

Using axioms on equality we can construct a q_0 for which we have (1.7).

(Case 1.20) is similar to (Case 1.19) and we omit it.

(Case 1.21) In this case a is:

$$a_0, \forall(x, y; IHx, IHy \to a_1), \quad \forall(x, y; IHx, IHy \to a_2) \to b$$

and a^* is:

$$a_0^*, \forall(x, y; IHx^*, IHy^* \to a_1^*), \quad \forall(x, y; IHx^*, IHy^* \to a_2^*) \to b^*$$

Then by modifying p_0 suitably we can construct a q_0 for which we have (1.6). (We omit the details of the construction.)

We thus found a q_0 for which we have (1.6) or (1.7). In case we have (1.7) we are done, so let us assume that we have (1.6). Then by Theorem 2.1 (with FOT replacing Univ) we obtain:

$$\vdash_{\text{FOT}} \text{proves}[q_0, \text{axiom}[a], \mathbb{BSA}] \tag{1.8}$$

Since a is a wff in $\mathbb{SA}_0$ and E is a wff_list in $\mathbb{SA}_0$ we have:

$$\vdash_{\text{FOT}} \text{wff}[a^*, \mathbb{BSA}] \tag{1.9}$$

$$\vdash_{\text{FOT}} \text{wff\_list}[E^*, \mathbb{BSA}] \tag{1.10}$$

By (1.8)-(1.10) we have:

$$\vdash_{\mathrm{FOT}} \mathrm{pr}[\mathbb{BSA}, E^*, [a^*, \mathrm{axiom}[q_0]], a^*]$$

This completes the proof of the proposition for (Case 1).

(Case 2) is easy and we omit it.

(Case 3) In this case we have for some P and A:

$$p \equiv [a, \mathrm{and\_I}[.P]] \tag{3.1}$$

$$a \equiv \mathrm{and}[.A] \tag{3.2}$$

and

$$\vdash_{\mathrm{FOT}} \mathrm{lpr}[\mathbb{SA}_0, E, P, A] \tag{3.3}$$

From (3.2) we have:

$$a^* \equiv \mathrm{and}[.A^*] \tag{3.4}$$

By (3.1) we see that P is a descendant of p. Hence we may apply the induction hypothesis to (3.3) and obtain:

$$\vdash_{\mathrm{FOT}} \mathrm{lpr}[\mathbb{BSA}, E^*, Q, A^*]$$

for some Q. From (3.4) and (3.5) we can conclude:

$$\vdash_{\mathrm{FOT}} \mathrm{pr}[\mathbb{BSA}, E^*, [a^*, \mathrm{and\_I}[.Q]], a^*] \tag{3.6}$$

We omit the proof of the remaining cases since they can be proved similarly as above. We have thus proved the first conjunt of our target proposition. We leave the proof of the remaining three conjunts as an exercise for the reader. By these arguments we have shown that $\mathbb{SA}_0$ is a conservative extension of $\mathbb{BSA}$.

We next show that $\mathbb{SA}$ is conservative over $\mathbb{SA}_0$. Since the wffs in $\mathbb{SA}$ and $\mathbb{SA}_0$ are the same, we have only to show that:

$$\mathbb{SA} \vdash a \Longrightarrow \mathbb{SA}_0 \vdash a$$

To show this it is sufficient to prove that:

$$\vdash_{\mathrm{FOT}} \mathrm{pr}[\mathbb{SA}, E, p, a] \Longrightarrow \vdash_{\mathrm{FOT}} \mathrm{pr}[\mathbb{SA}_0, E, q, a] \text{ for some } q$$

In fact, as in the previous proof, we must prove a stronger proposition involving lpr, upr and eq_pr by the $<$-induction on p, but since the essential point lies in the proof of the above proposition (which is a conjunct of the stronger proposition) let us pretend that this is our target. So assume that:

$$\vdash_{\mathrm{FOT}} \mathrm{pr}[\mathbb{SA}, E, p, a]$$

As before, we have 14 cases to consider, but the crucial case is (Case 1). Now (Case 1) produces 21 subcases depending on which axiom is actually used and the only nontrivial case is (Case 1.22) where (formal) $<$-induction is applied. In this case a is a wff in $\mathbb{SA}$ and we have for some p_0, w, z, $a1$, b, $a0$, t:

$$p \equiv [a, \text{axiom}[p_0]] \tag{1}$$

$$a \equiv \forall(z; \forall(w; w < z \to a1) \to a0) \to b. \tag{2}$$

$$\vdash_{\mathbb{SA}} \text{pure\_variable}[z] \tag{3}$$

$$\vdash_{\mathbb{SA}} \text{pure\_variable}[w] \tag{4}$$

$$\vdash_{\mathbb{SA}} \text{ne}[z, w] \tag{5}$$

$$\vdash_{\mathbb{SA}} \text{new}[w, a0] \tag{6}$$

$$\vdash_{\mathbb{SA}} \text{wff}[a0] \tag{7}$$

$$\vdash_{\mathbb{SA}} \text{term}[t] \tag{8}$$

$$\vdash_{\mathbb{SA}} \text{substl}[w, z, a0, a1] \tag{9}$$

and

$$\vdash_{\mathbb{SA}} \text{substl}[t, z, a0, b] \tag{10}$$

We show that a is provable in $\mathbb{SA}_0$ by using induction axiom in $\mathbb{SA}_0$. Let us choose two distinct pure variables x and y which are new to $a1$ and put:

$$A \equiv \forall(z; \forall(w; w < z \to a1) \to a0) \to \forall(w; w < z \to a1)$$

$$T \equiv [t\,.\,t]$$

$$B \equiv A_z[T]$$

$$A0 \equiv A_z[*]$$

$$IHx \equiv A_z[x]$$

$$IHy \equiv A_z[y]$$

$$A1 \equiv A_z[[x\,.\,y]]$$

$$A2 \equiv A_z[(:x\,.\,y)]$$

Then we have:

$$\vdash_{\mathbb{SA}_0} \text{axiom}[C] \tag{11}$$

where

$$C \equiv A0, \forall(x, y; IHx, IHy \to A1), \forall(x, y; IHx, IHy \to A2) \to B \tag{12}$$

Thus for some p_0 we have:

$$\vdash_{\text{FOT}} \text{pr}[\mathbb{SA}_0, [\,], [C, \text{axiom}[p_0]], C] \tag{13}$$

This means:

$$\mathbb{SA}_0 \vdash C \tag{14}$$

By direct computation we have:

$$A0 \equiv \forall(z;\ \forall(w;\ w < z \to a1) \to a0) \to \forall(w;\ w < * \to a1) \tag{15}$$

Now to prove the wff a in $\mathbb{SA}_0$ let us work in $\mathbb{SA}_0$. We first prove $A0$. Assume that:

$$\forall(z;\ \forall(w;\ w < z \to a1) \to a0) \tag{A1}$$

(In fact, this assumption will not be used.) Assume further that:

$$w < * \tag{A2}$$

Then we get $a1$ by axiom 8 and $(\bot E)$. By discharging (A2) and then applying $(\forall I)$ we get:

$$\forall(w;\ w < * \to a1)$$

We then get (15) by discharging (A1).

Next we prove:

$$\forall(x,\ y;\ IHx,\ IHy \to A1) \tag{16}$$

Assume IHx, that is:

$$\forall(z;\ \forall(w;\ w < z \to a1) \to a0) \to \forall(w;\ w < x \to a1) \tag{A3}$$

and IHy, that is:

$$\forall(z;\ \forall(w;\ w < z \to a1) \to a0) \to \forall(w;\ w < y \to a1) \tag{A4}$$

Now to prove $A1$ we assume (A1) as before. Then we have:

$$\forall(w;\ w < x \to a1) \tag{C1}$$

and

$$\forall(w;\ w < y \to a1) \tag{C2}$$

By specializing z to x and y in (A1) we get

$$\forall(w;\ w < x \to a1) \to a0_z[x] \tag{C3}$$

and

$$\forall(w;\ w < y \to a1) \to a0_z[y] \tag{C4}$$

Using (C1)–(C4) we have:

$$a0_z[x] \tag{C5}$$

and

$$a0_z[y] \tag{C6}$$

Our goal is the conclusion of $A1$, that is:

$$\forall(w;\ w < [x\,.\,y] \to a1) \tag{G1}$$

To prove this assume:

$$w < [x\,.\,y] \tag{A5}$$

Then by axiom 14 we have 4 cases to consider. We will prove $a1$ in all cases.

(Case 1) In this case we have $w = x$. Then by (C5) and $(=subst)$ we have $a0_z[w]$, that is, $a1$.

(Case 2) In this case we have $w < x$. By specializing (C1) to w we have $a1$.

(Case 3) and (Case 4) can be proved similarly.

From these we can deduce (G1). By discharging (A1) we get (16).

We can also prove:

$$\forall(x,\ y;\ IHx,\ IHy \to A2)$$

quite similarly.

Now in view of (12) we may conclude B that is:

$$\forall(z;\ \forall(w;\ w < z \to a1) \to a0) \to \forall(w;\ w < [t\,.\,t] \to a1) \tag{17}$$

On the other hand noting that $t < [t\,.\,t]$ and $b \equiv a1_w[t]$ we have:

$$\forall(w;\ w < [t\,.\,t] \to a1) \to b \tag{18}$$

By (17) and (18) we get a. We have thus proved the most crucial and the only nontrivial case. □

5.3. modification on FOT

The formal system FOT we introduced in Section 4.2 provided a general framework for defining arbitrary first order theories. We defined two first order theories, namely, BSA and SA in this framework and showed that these theories are equivalent. In the rest of the paper we will formally work in SA and will develop elementary mathematics in it. For this purpose, we find it better to modify FOT so that this enterprise will become easier to carry out. Roughly speaking, we will modify FOT so that all the theories defined by it will contain the language and axioms of SA. This means that in the modified FOT the empty list [ ] will define the axiom system SA. This modification prevents us from treating the systems not containing SA, however, for us this is not a

problem since we are not interested in such systems at the moment. On the other hand, the modified FOT can be used to define any extension of SA whose language and axioms are recursively enumerable.

We now modify FOT as follows. We replace the definitions of the predicates 'term', 'wff' and 'axiom' by the following three sexps *Term*, *Wff* and *Axiom*. We also define the predicate 'sa_axiom' by the fourth sexp *Sa_axiom*. From now on we will call thus obtained formal system as FOT.

```
+ term
      | x, S
            - variable[x]
      | *, S
      | [s.t], S
            - term[s, S]
            - term[t, S]
      | '(: /s./t), S
            - term[s, S]
            - term[t, S]
      | (: apply.[Fun.T]), S
            - symbol[Fun]
            - term_list[T, S]
            - length[T, Arity]
            - theorem[function[[Fun.Arity]], S]
      | (: quote.t), S
      | (: *.t), S
      ;
+ wff
      | eq[s, t], S
            - term[s, S]
            - term[t, S]
      | lt[s, t], S
            - term[s, S]
            - term[t, S]
      | [Prd.T], S
            - admissible[Prd]
            - term_list[T, S]
```

```
            – length[T, Arity]
            – theorem[predicate[[Prd . Arity]], S]
      | and [ . A], S
            – wff_list[A, S]
      | or[ . A], S
            – wff_list[A, S]
      | imply[A, b], S
            – wff_list[A, S]
            – wff[b, S]
      | all[(: abs . [X, a])], S
            – pure_variable_list[X]
            – wff[a, S]
      | ex[(: abs . [X, a])], S
            – pure_variable_list[X]
            – wff[a, S]
      ;
+ axiom
      | a, S
            – sa_axiom[a, S]
      | a, S
            – proves[p, axiom[a], S]
            – wff[a, S]
      ;
+ sa_axiom
      | 'A(s, t; [s . t] ≠ *), S % 1 %
      | 'A(s, t; (: s . t) ≠ *), S % 2 %
      | 'A(s, t, u, v; [s . t] ≠ (: u . v)), S % 3 %
      | 'A(s, t, u, v; [s . t] = [u . v] → s = u), S % 4 %
      | 'A(s, t, u, v; [s . t] = [u . v] → t = v), S % 5 %
      | 'A(s, t, u, v; (: s . t) = (: u . v) → s = u), S % 6 %
      | 'A(s, t, u, v; (: s . t) = (: u . v) → t = v), S % 7 %
      | 'A(r; ¬(r < *)), S % 8 %
      | 'A(r, s, t; ¬(r < (: s . t))), S % 9 %
      | 'A(s, t; s < [s, t]), S % 10 %
      | 'A(s, t; t < [s . t]), S % 11 %
      | 'A(r, s, t; r < s → r < [s . t]), S % 12 %
```

```
| '∀(r, s, t; r < t → r < [s.t]), S % 13 %
| '∀(r, s, t; r < [s.t] →
      r = s ∨ r < s ∨ r = t ∨ r < t), S % 14 %
| '((quote.*) = *), S % 15 %
| `((quote.[/s./t]) = [(quote./s).(quote./t)]), S % 16 %
| `((quote.(/s./t)) = (:(quote./s).(quote./t))), S % 17 %
| '((*.*) = (:*.*)), S % 18 %
| `((*./s) = (:*./u), (*./t) = (:*./v) →
      (*.[/s./t]) = (:*.[/u./v])), S % 19 %
| `((*./s) = (:*./u), (*./t) = (:*./v) →
      (*.(/s./t)) = (:*.(:/u./v))), S % 20 %
| `(/a0, ∀(/x, /y; /IHx, IHy → /a1),
      ∀(/x, /y; /IHx, /IHy → /a2) → /b), S % 21 %
      – pure_variable[x]
      – pure_variable[y]
      – pure_variable[z]
      – ne[x, y]
      – wff[a, S]
      – new[x, a]
      – new[y, a]
      – term[t, S]
      – substl[t, z, a, b]
      – substl[*, z, a, a0]
      – substl[x, z, a, IHx]
      – substl[y, z, a, IHy]
      – substl[[x.y], z, a, a1]
      – substl[`(:/x./y), z, a, a2]
| `(∀(/z; ∀(/w; /w < /z → /a1) → /a) → /b), S % 22 %
      – pure_variable[z]
      – pure_variable[w]
      – ne[z, w]
      – wff[a, S]
      – new[w, a]
      – term[t, S]
      – substl[w, z, a, a1]
      – substl[t, z, a, b]
```

;

Let us refer to the old FOT as FOT_{old}. Then it is easy to see that:

$$\vdash_{\mathrm{FOT}_{old}} \mathrm{pr}[\mathbb{SA}, E, p, a] \text{ for some } p$$
$$\Longleftrightarrow \vdash_{\mathrm{FOT}} \mathrm{pr}[[\ ], E, p, a] \text{ for some } q$$

We therefore redefine SA as the empty list [ ]. The notation

$$S \vdash a$$

is an abbreviation of

$$\vdash_{\mathrm{FOT}} \mathrm{pr}[S, [\ ], p, a] \text{ for some } p$$

where FOT now refers to the modified one.

§6. Mathematics in SA

In this section we will formally work in the axiom system SA and will develop elementary mathematics within it.

6.1. elementary properties of SA

To begin with, let us prove some simple theorems in SA. We will display a formal theorem in SA in the following form.

Thm 6.1.1. $s \leqq t \leftrightarrow s < [t.t]$

Proof. Although this is a formal theorem, we give its proof informally. We prove the $\leftarrow$ part first. Assume $s < [t.t]$. Then by axiom 14, we have

$$s=t \vee s<t \vee s=t \vee s<t$$

From this, by logic, we have $s<t \vee s=t$ as desired. For the $\rightarrow$ part, assume $s \leqq t$, that is $s<t \vee s=t$. Then we have two cases.

(Case 1) $s<t$. In this case we have $s<[t.t]$ by axiom 12.

(Case 2) $s=t$. By axiom 10, we have $t<[t.t]$. Since $s=t$, we have $s<[t.t]$. □

We give one more example that uses ordinary induction.

Thm 6.1.2. $s < t, t < u \rightarrow s < u$

Proof. We prove this theorem by induction on u.

(Basis) Assume $s < t$ and $t < *$. From the second assumption we can derive a contradiction by axiom 8.

(Step cons) We prove the theorem for [u1 . u2], assuming the theorem for u1 and u2. So assume s < t and t < [u1 . u2]. By the second assumption and axiom 14, we have four cases to consider.

(Case 1) t = u1. In this case we have s < u1 by the first assumption. From this by axiom 12 we have s < [u1 . u2].
(Case 2) t < u1. In this case we have s < u1 by induction hypothesis. From this we have s < [u1 . u2] by axiom 12.
Cases 3 and 4 are proved similarly.

(Step snoc) We prove the theorem for (: u1 . u2). Assume s < t and t < (: u1 . u2). By the second assumption and axiom 9, we get a contradiction. From this we can deduce the desired result. □

6.2. abstract

In SA mathematical concepts are expressed in terms of wffs. For instance the concept of an atom is expressed in SA by the wff ∃(x, y; z = (: x . y)) which means that z is an atom. As such mathematical concepts become more sophisticated the wff representing these concepts also become very complicated. We therefore need a systematic way of giving names to wffs in SA. We define the concept of an *abstract* for this purpose. Let $x_1,\ldots, x_n$ be a sequence of distinct pure variables and let d be a designator, i.e., wff or a term. Then the sexp:

$$\text{abstract}[(\text{abs} . [[x_1,\ldots, x_n], d])]$$

will be called an abstract of arity n. This sexp will also be written as:

$$\lambda(x_1,\ldots, x_n; d)$$

Let A be an abstract of the above form and let $t_1,\ldots, t_n$ be a sequence of sexps. Then

$$A(t_1,\ldots, t_n)$$

will denote the sexp:

$$d_{x_1,\ldots,x_n}\ [\![t_1,\ldots, t_n]\!]$$

It is a wff (term) if $t_1,\ldots, t_n$ are terms and d is a wff (term, resp.).

Example 6.1.

$$\lambda(z; \exists(x, y; z = (: x . y)))(\text{apple}) \equiv \exists(x, y; \text{apple} = (: x . y)) \quad \square$$

An abstract A is called *closed* if $A(*,\ldots, *)$ is closed. A closed abstract will also

be called a *predicate* if it is abstracted from a wff. We will allow to give a name to a closed abstract. We will use strings of alphanumeric characters whose first characters are uppercase letters as names of abstracts. For instance, if we wish to assign a name 'Atom' to the abstract:

$$\lambda(z; \exists(x, y; z = (: x . y)))$$

we do it as follows:

Def 6.1.1. $\mathrm{Atom}(z) = \exists(x, y; z = (: x . y))$

Similarly we define an abstract (whose name is 'Mole') by:

Def 6.1.2. $\mathrm{Mole}(z) = \exists(x, y; z = [x . y])$

The general format of the definition of an abstract is as follows. Let $\mathcal{d}$ be a designator and let $x_1,\ldots, x_n$ be a sequence of distinct pure variables. Then:

Def n *Name* $(x_1,\ldots, x_n) = \mathcal{d}$

gives the name *Name* to the abstract $\lambda(x_1,\ldots, x_n; \mathcal{d})$. (This abstract must be closed.)

Let us define one more abstract.

Def 6.1.3. $\mathrm{Null}(z) = (z = *)$

Using these abstracts we can state some simple theorems whose proofs we do not give here.

Thm 6.2.1. $\mathrm{Null}(x) \vee \mathrm{Mole}(x) \vee \mathrm{Atom}(x)$

Thm 6.2.2. $\mathrm{Null}(x), \mathrm{Mole}(x) \rightarrow \bot$

Thm 6.2.3. $\mathrm{Null}(x), \mathrm{Atom}(x) \rightarrow \bot$

Thm 6.2.4. $\mathrm{Mole}(x), \mathrm{Atom}(x) \rightarrow \bot$

These theorems are useful in proving the following theorems.

Thm 6.3.1. $x = y \vee x \neq y$

Thm 6.3.2. $x < y \vee \neg(x < y)$

We will also allow to give a name to a specific closed term in the following form:

Def 6.2.1. FOT = 'FOT

By this definition, FOT will denote the term 'FOT. This definition also has the effect of making FOT a reserved string, thereby prohibiting the use of FOT as a variable. Let us give one more definition.

Def 6.2.2. SA = SA

This definition simply makes SA a synonym for $*$.

6.3. representation of Univ in SA

In order to formalize in SA what we have been doing informally in this paper, we must be able to formally define predicates whose informal counterparts have been given by inductive definitions. In view of Section 2, this will be accomplished if we can define the concept of a formal system in SA, or equivalently, if we can define the predicate *proves* in SA. By Theorem 2.1 we can reduce this problem to the problem of representing the single formal system Univ in SA. We can solve this problem by defining an abstract Univ_tree such that Univ_tree(T) means that T is a proof in the formal system Univ. (For technical reasons, proof trees represented by Univ_tree are slightly different from the actual proof trees in Univ.)

Let us now define Univ_tree. We first define Pnode as auxiliary concept.

Def 6.3.1. Pnode(t) = $\exists$(x, y; t = [(: x) . y]))

Using Pnode, Univ_tree is defined as follows.

Def 6.3.2.

```
Univ_tree(T)
= Pnode(T) ∧
∀(q ≦ T; Pnode(q) →
      ∃(u, v; q = [(: ne[*, [u . v]])])
      ∨∃(u, v; q = [(: ne[*, (: u . v)])])
      ∨∃(s, t; q = [(: ne[[s . t], *])])
      ∨∃(s, t; q = [(: ne[(: s . t), *])])
      ∨∃(s, t, u, v; q = [(: ne[[s . t], (: u . v)])])
      ∨∃(s, t, u, v; q = [(: ne[(: s . t), [u . v]])])
      ∨∃(s  t, u, v, Q; q = [(: ne[[s . t], [u . v]]), [(: ne[s, u]) . Q]])
      ∨∃(s, t, u, v, Q; q = [(: ne[[s . t], [u . v]]), [(: ne[t, v]) . Q]])
      ∨∃(s, t, u, v, Q;
            q = [(: ne[(: s . t), (: u . v)]), [(: ne[s, u]) . Q]])
      ∨∃(s, t, u, v, Q;
            q = [(: ne[(: s . t), (: u . v)]), [(: ne[t, v]) . Q]])
      ∨∃(x, v, L; q = [(: assoc[x, [[x . v] . L], v])])
      ∨∃(x, y, w, L, v, Q1, Q2;
```

```
                q = [(: assoc[x, [[y . w] . L], v]),
                     [(: ne[x, y]) . Q1],
                     [(: assoc[x, L, v]) . Q2]])
            ∨ ∃(v, L; q = [(: get[*, [v . L], v])])
            ∨ ∃(i, w, L, v, Q;
                q = [(: get[[* . i], [w . L], v]), [(: get[i, L, v]) . Q]])
            ∨ ∃(t, E, v, Q; q = [(: eval[(: var . t), E, v]),
                     [(: assoc[(: var . t), E, v]) . Q]])
            ∨ ∃(E; q = [(: eval [*, E, *])])
            ∨ ∃(s, t, E, u, v, Q1, Q2;
                q = [(: eval[[s . t], E, [u . v]]),
                     [(: eval[s, E, u]) . Q1],
                     [(: eval[t, E, v]) . Q2]])
            ∨ ∃(s, t, E, u, v, Q1, Q2;
                q = [(: eval[(snoc . [s, t]), E, (: u . v)]),
                     [(: eval[s, E, u]) . Q1],
                     [(: eval[t, E, v]) . Q2]])
            ∨ ∃(t, E; q = [(: eval[(: * . t), E, (: * . t)])])
            ∨ ∃(t, E; q = [(: eval[(: quote . t), E, t])])
            ∨ ∃(Prd, FS, R, i, c, C, Env, a, A, P, Q1, Q2, Q3, Q4, Q5;
                q = [(: proves[[[Prd, i, Env] . P], [Prd . a], FS]),
                     [(: assoc[Prd, FS, R]) . Q1],
                     [(: get[i, R, [c . C]]) . Q2],
                     [(: eval[c, Env, a]) . Q3],
                     [(: eval[C, Env, A]) . Q4],
                     [(: lproves[P, A, FS]) . Q5]])
            ∨ ∃(FS; q = [(: lproves[[ ], [ ], FS])])
            ∨ ∃(p, P, a, A, FS, Q1, Q2;
                q = [(: lproves[[p . P], [a . A], FS]),
                     [(: proves[p, a, FS]) . Q1],
                     [(: lproves[P, A, FS]) . Q2]])
            ∨ ∃(a, FS, p, Q; q = [(: theorem[a, FS]),
                     [(: proves[p, a, FS]) . Q]])
)
```

Let us examine the basic structure of a sexp defined by Univ_tree. We first prove the following lemma.

Thm 6.4.1. Univ_tree(T), Pnode(q), q < T → Univ_tree(q)

Proof. Assume that:

$$\text{Univ\_tree}(T) \tag{1}$$

$$\text{Pnode}(q) \tag{2}$$

and

$$q < T \tag{3}$$

By the definition of Univ_tree we have:

$$\text{Univ\_tree}(T) \equiv \text{Pnode}(T) \wedge \forall(q \leqq T;\ \text{Pnode}(q) \rightarrow \mathcal{a})$$

where $\mathcal{a}$ is a disjuction of 24 wffs. Hence by (1) we have:

$$\forall(q \leqq T;\ \text{Pnode}(q) \rightarrow \mathcal{a}) \tag{4}$$

By (3) and (4) using the transitivity of $<$ we have:

$$\forall(q \leqq q;\ \text{Pnode}(q) \rightarrow \mathcal{a}) \tag{5}$$

(Recall that (5) is an abbreviation of $\forall(q;\ q \leqq \sharp q \rightarrow (\text{Pnode}(q) \rightarrow \mathcal{a}))$.) From (2) and (5) we get Univ_tree(q). □

The lemma we just proved is useful in proving the following theorem which characterizes Univ_tree. To state the theorem we introduce the following abstract.

Def 6.4.1. Der(T, c) = Univ_tree(T) ∧ ∃(Q; T = [(: c) . Q])

Thm 6.4.2.

$$\begin{aligned}
&\text{Univ_tree}(T) \leftrightarrow \\
&\quad \exists(u, v;\ T = [(:\ \text{ne}[*, [u\,.\,v]])]) \\
&\quad \vee \exists(u, v;\ T = [(:\ \text{ne}[*, (:\ u\,.\,v)])]) \\
&\quad \vee \exists(s, t;\ T = [(:\ \text{ne}[[s\,.\,t], *])]) \\
&\quad \vee \exists(s, t;\ T = [(:\ \text{ne}[(:\ s\,.\,t), *])]) \\
&\quad \vee \exists(s, t, u, v;\ T = [(:\ \text{ne}[[s\,.\,t], (:\ u\,.\,v)])]) \\
&\quad \vee \exists(s, t, u, v;\ T = [(:\ \text{ne}[(:\ s\,.\,t), [u\,.\,v]])]) \\
&\quad \vee \exists(s, t, u, v, T1;\ \text{Der}(T1, \text{ne}[s, u]) \wedge \\
&\qquad\quad T = [(:\ \text{ne}[[s\,.\,t], [u\,.\,v]]), T1]) \\
&\quad \vee \exists(s, t, u, v, T1;\ \text{Der}(T1, \text{ne}[t, v]) \wedge \\
&\qquad\quad T = [(:\ \text{ne}[[s\,.\,t], [u\,.\,v]]), T1]) \\
&\quad \vee \exists(s, t, u, v, T1;\ \text{Der}(T1, \text{ne}[s, u]) \wedge \\
&\qquad\quad T = [(:\ \text{ne}[(:\ s\,.\,t), (:\ u\,.\,v)]), T1]) \\
&\quad \vee \exists(s, t, u, v, T1;\ \text{Der}(T1, \text{ne}[t, v]) \wedge
\end{aligned}$$

$$
\begin{aligned}
&\quad T = [(: \mathrm{ne}[(: s . t), (: u . v)]), T1]) \\
&\vee \exists(x, v, L;\ T = [(: \mathrm{assoc}[x, [[x . v] . L], v])]) \\
&\vee \exists(x, y, w, L, v, T1, T2; \\
&\qquad \mathrm{Der}(T1, \mathrm{ne}[x, y]) \wedge \mathrm{Der}(T2, \mathrm{assoc}[x, L, v]) \\
&\qquad \wedge T = [(: \mathrm{assoc}[x, [[y . w] . L], v]), T1, T2]) \\
&\vee \exists(v, L;\ T = [(: \mathrm{get}[*, [v . L], v])]) \\
&\vee \exists(i, w, L, v, T1;\ \mathrm{Der}(T1, \mathrm{get}[i, L, v]) \\
&\qquad \wedge T = [(: \mathrm{get}[[* . i], [w . L], v]), T1]) \\
&\vee \exists(t, E, v, T1;\ \mathrm{Der}(T1, \mathrm{assoc}[(: \mathrm{var} . t), E, v]) \\
&\qquad \wedge T = [(: \mathrm{eval}[(: \mathrm{var} . t), E, v]), T1]) \\
&\vee \exists(E;\ T = [(: \mathrm{eval}[*, E, *])]) \\
&\vee \exists(s, t, E, u, v, T1, T2; \\
&\qquad \mathrm{Der}(T1, \mathrm{eval}[s, E, u]) \wedge \mathrm{Der}(T2, \mathrm{eval}[t, E, v]) \\
&\qquad \wedge T = [(: \mathrm{eval}[[s, t], E, [u . v]]), T1, T2]) \\
&\vee \exists(s, t, E, u, v, T1, T2; \\
&\qquad \mathrm{Der}(T1, \mathrm{eval}[s, E, u]) \wedge \mathrm{Der}(T2, \mathrm{eval}[t, E, v]) \\
&\qquad \wedge T = [(: \mathrm{eval}[(: \mathrm{snoc} . [s, t]), E, (: u . v)]), T1, T2]) \\
&\vee \exists(t, E;\ T = [(: \mathrm{eval}[(: * . t), E, (: * . t)])]) \\
&\vee \exists(t, E;\ T = [(: \mathrm{eval}[(: \mathrm{quote} . t), E, t])]) \\
&\vee \exists(\mathrm{Prd}, \mathrm{FS}, R, i, c, C, \mathrm{Env}, a, A, P, T1, T2\ T3, T4, T5; \\
&\qquad \mathrm{Der}(T1, \mathrm{assoc}[\mathrm{Prd}, \mathrm{FS}, R]) \wedge \mathrm{Der}(T2, \mathrm{get}[i, R, [c . C]]) \\
&\qquad \wedge \mathrm{Der}(T3, \mathrm{eval}[c, \mathrm{Env}, a]) \wedge \mathrm{Der}(T4, \mathrm{eval}[C, \mathrm{Env}, A]) \\
&\qquad \wedge \mathrm{Der}(T5, \mathrm{lproves}[P, A, \mathrm{FS}]) \\
&\qquad \wedge T = [(: \mathrm{proves}[[[\mathrm{Prd}, i, \mathrm{Env}] . P], [\mathrm{Prd} . a], \mathrm{FS}]), \\
&\qquad\qquad T1, T2, T3, T4, T5]) \\
&\vee \exists(\mathrm{FS};\ T = [(: \mathrm{lproves}[[\ ], [\ ], \mathrm{FS}])]) \\
&\vee \exists(p, P, a, A, \mathrm{FS}, T1, T2; \\
&\qquad \mathrm{Der}(T1, \mathrm{proves}[p, a, \mathrm{FS}]) \wedge \mathrm{Der}(T2, \mathrm{lproves}[P, A, \mathrm{FS}]) \\
&\qquad \wedge T = [(: \mathrm{lproves}[[p . P], [a . A], \mathrm{FS}]), T1, T2]) \\
&\vee \exists(a, \mathrm{FS}, p, T1;\ \mathrm{Der}(T1, \mathrm{proves}[p, a, \mathrm{FS}]) \\
&\qquad \wedge T = [(: \mathrm{theorem}[a, \mathrm{FS}]), T1])
\end{aligned}
$$

Proof. We first prove the $\rightarrow$ part. Assume that Univ_tree(T). Then we have Pnode(T) and

$$\forall(q \leqq T;\ \mathrm{Pnode}(q) \rightarrow \mathscr{a}) \tag{1}$$

By specializing (1) to T and using the fact that Pnode(T), we get $\mathscr{b}$ where $\mathscr{b}$ is

$\mathfrak{a}_q[\![\mathrm{T}]\!]$. Since $\mathfrak{b}$ is a disjunction of 24 wffs, we let $\mathfrak{b}_i$ be the i-th disjunct of $\mathfrak{b}$. We now have 24 cases to consider where in the i-th case we may assume $\mathfrak{b}_i$. Since these cases may be treated rather similarly we only consider the case 17 as a typical case.

(Case 17) In this case we may assume $\mathfrak{b}_{17}$, that is:

$$\begin{aligned}&\exists(\mathrm{s},\ \mathrm{t},\ \mathrm{E},\ \mathrm{u},\ \mathrm{v},\ \mathrm{Q1},\ \mathrm{Q2};\\ &\quad \mathrm{T} = [(\mathrm{:}\ \mathrm{eval}[[\mathrm{s}\,.\,\mathrm{t}],\ \mathrm{E},\ [\mathrm{u}\,.\,\mathrm{v}]]),\\ &\qquad [(\mathrm{:}\ \mathrm{eval}[\mathrm{s},\ \mathrm{E},\ \mathrm{u}])\,.\,\mathrm{Q1}],\\ &\qquad [(\mathrm{:}\ \mathrm{eval}[\mathrm{t},\ \mathrm{E},\ \mathrm{v}])\,.\,\mathrm{Q2}]])\end{aligned}$$

Let s, t, E, u, v, Q1 and Q2 be such that:

$$\begin{aligned}\mathrm{T} = [&(\mathrm{:}\ \mathrm{eval}[[\mathrm{s}\,.\,\mathrm{t}],\ \mathrm{E},\ [\mathrm{u}\,.\,\mathrm{v}]]),\\ &[(\mathrm{:}\ \mathrm{eval}[\mathrm{s},\ \mathrm{E},\ \mathrm{u}])\,.\,\mathrm{Q1}],\\ &[(\mathrm{:}\ \mathrm{eval}[\mathrm{t},\ \mathrm{E},\ \mathrm{v}])\,.\,\mathrm{Q2}]]\end{aligned} \tag{1}$$

We put

$$\begin{aligned}&c \equiv \mathrm{eval}[[\mathrm{s}\,.\,\mathrm{t}],\ \mathrm{E},\ [\mathrm{u}\,.\,\mathrm{v}]],\\ &T1 \equiv [(\mathrm{:}\ \mathrm{eval}[\mathrm{s},\ \mathrm{E},\ \mathrm{u}])\,.\,\mathrm{Q1}]\end{aligned}$$

and

$$T2 \equiv [(\mathrm{:}\ \mathrm{eval}[\mathrm{t},\ \mathrm{E},\ \mathrm{v}])\,.\,\mathrm{Q2}]$$

Then by (1) we have:

$$\mathrm{T} = [(\mathrm{:}\ c),\ T1,\ T2]. \tag{2}$$

Now it is easy to see that Pnode($T1$) and $T1 < \mathrm{T}$ hold. Then by Thm 6.4.1 we have:

$$\mathrm{Univ\_tree}(T1) \tag{3}$$

Similarly we have:

$$\mathrm{Univ\_tree}(T2) \tag{4}$$

From (3) and (4) we get:

$$\mathrm{Der}(T1,\ \mathrm{eval}[\mathrm{s},\ \mathrm{E},\ \mathrm{u}]) \tag{5}$$

and

$$\mathrm{Der}(T2,\ \mathrm{eval}[\mathrm{t},\ \mathrm{E},\ \mathrm{v}]) \tag{6}$$

Using (2), (5) and (6) we obtain:

$$\exists(s, t, E, u, v, T1, T2;$$
$$\quad \mathrm{Der}(T1, \mathrm{eval}[s, E, u]) \wedge \mathrm{Der}(T2, \mathrm{eval}[t, E, v])$$
$$\quad \wedge\ T = [(: \mathrm{eval}[[s\,.\,t], E, [u\,.\,v]]), T1, T2])$$

From this we obtain the desired result by $(\vee I)$.

We next prove the converse by the $<$-induction on T. We have 24 cases to consider but we only treat the following case:

(Case 17) In this case we can take s, t, E, u, v, T1 and T2 for which we have:

$$\mathrm{Der}(T1, \mathrm{eval}[s, E, u]) \tag{1}$$

$$\mathrm{Der}(T2, \mathrm{eval}[t, E, v]) \tag{2}$$

and

$$T = [(: \mathrm{eval}[[s, t], E, [u\,.\,v]]), T1, T2] \tag{3}$$

By (3) we have:

$$\mathrm{Pnode}(T) \tag{4}$$

Now take any q such that $q \leqq T$ and Pnode(q). Then by (3), using the rules concerning $<$, we have:

$$q = T \vee q \leqq T1 \vee q \leqq T2$$

We have therefore three cases. First assume $q = T$. Then by (1), (2) and (3) we have:

$$\exists(s, t, E, u, v, Q1, Q2;$$
$$\quad q = [(: \mathrm{eval}[[s\,.\,t], E, [u\,.\,v]]),$$
$$\qquad [(: \mathrm{eval}[s, E, u])\,.\,Q1],$$
$$\qquad [(: \mathrm{eval}[t, E, v])\,.\,Q2]])$$

From this we have $\mathfrak{a}$ by $(\vee I)$. Next assume $q \leqq T1$. In this case we can easily derive $\mathfrak{a}$ from (1). In case $q \leqq T2$, we obtain $\mathfrak{a}$ similarly by using (2). We therefore have:

$$\forall(q \leqq T;\ \mathrm{Pnode}(q) \to \mathfrak{a}) \tag{5}$$

From (4) and (5) we obtain Univ_tree(T). □

By Thm 6.4.2 we see that a Univ_tree has a structure very similar to a proof in the formal system Umiv.

6.4. formal systems in SA

Using the predicate Univ_tree we can now define the predicate Proves and related predicates as follows. By these predicates we can describe formal systems in SA.

Def 6.5.1. $\mathrm{Ne}(x, y) = \exists(T; \mathrm{Der}(T, \mathrm{ne}[x, y]))$

Def 6.5.2. $\mathrm{Assoc}(x, L, v) = \exists(T; \mathrm{Der}(T, \mathrm{assoc}[x, L, v]))$

Def 6.5.3. $\mathrm{Get}(i, L, v) = \exists(T; \mathrm{Der}(T, \mathrm{get}[i, L, v]))$

Def 6.5.4. $\mathrm{Eval}(t, \mathrm{Env}, v) = \exists(T; \mathrm{Der}(T, \mathrm{eval}[t, \mathrm{Env}, v]))$

Def 6.5.5. $\mathrm{Proves}(p, a, FS) = \exists(T; \mathrm{Der}(T, \mathrm{proves}[p, a, FS]))$

Def 6.5.6. $\mathrm{Lproves}(P, A, FS) = \exists(T; \mathrm{Der}(T, \mathrm{lproves}[P, A, FS]))$

Def 6.5.7. $\mathrm{Theorem}(a, FS) = \exists(T; \mathrm{Der}(T, \mathrm{theorem}[a, FS]))$

The following theorems show that these predicates have the desired properties.

Thm 6.5.1. $\mathrm{Ne}(x, y) \leftrightarrow x \neq y$

Thm 6.5.2.

$$
\begin{aligned}
&\mathrm{Ne}(x, y) \leftrightarrow \\
&\quad \exists(u, v; x = * \wedge y = [u.v]) \\
&\quad \vee \exists(u, v; x = * \wedge y = (:u.v)) \\
&\quad \vee \exists(s, t; x = [s.t] \wedge y = *) \\
&\quad \vee \exists(s, t; x = (:s.t) \wedge y = *) \\
&\quad \vee \exists(s, t, u, v; x = [s.t] \wedge y = (:u.v)) \\
&\quad \vee \exists(s, t, u, v; x = (:s.t) \wedge y = [u.v]) \\
&\quad \vee \exists(s, t, u, v; x = [s.t] \wedge y = [u.v] \wedge \mathrm{Ne}(s, u)) \\
&\quad \vee \exists(s, t, u, v; x = [s.t] \wedge y = [u.v] \wedge \mathrm{Ne}(t, v)) \\
&\quad \vee \exists(s, t, u, v; x = (:s.t) \wedge y = (:u.v) \wedge \mathrm{Ne}(s, u)) \\
&\quad \vee \exists(s, t, u, v; x = (:s.t) \wedge y = (:u.v) \wedge \mathrm{Ne}(t, v))
\end{aligned}
$$

Thm 6.5.3.

$$
\begin{aligned}
&\mathrm{Assoc}(x, L, v) \leftrightarrow \\
&\quad \exists(L; \#L = [[x.v].L]) \\
&\quad \vee \exists(y, w, L; \#L = [[y.w].L] \wedge \mathrm{Ne}(x, y) \wedge \mathrm{Assoc}(x, L, v))
\end{aligned}
$$

Thm 6.5.4.

$$
\begin{aligned}
&\mathrm{Get}(i, L, v) \leftrightarrow \\
&\quad \exists(L; i = * \wedge \#L = [v.L])
\end{aligned}
$$

$$\vee\exists(i, w, L; \#i = [*.i] \wedge \#L = [w.L] \wedge \mathrm{Get}(i, L, v))$$

Thm 6.5.5.

$$\begin{aligned}&\mathrm{Eval}(t, \mathrm{Env}, v)\leftrightarrow\\ &\quad\exists(t; \#t = (:\mathrm{var}.t) \wedge v = (:\mathrm{var}.t))\\ &\quad\vee(t = * \wedge v = *)\\ &\quad\vee\exists(s, t, u, v; \#t = [s.t] \wedge \#v = [u.v]\\ &\qquad\wedge \mathrm{Eval}(s, \mathrm{Env}, u) \wedge \mathrm{Eval}(t, \mathrm{Env}, v))\\ &\quad\vee\exists(s, t, u, v; \#t = (:\mathrm{snoc}.[s, t]) \wedge \#v = (:u.v)\\ &\qquad\wedge \mathrm{Eval}(s, \mathrm{Env}, u) \wedge \mathrm{Eval}(t, \mathrm{Env}, v))\\ &\quad\vee\exists(t; \#t = (:*.t) \wedge v = (:*.t))\\ &\quad\vee\exists(t; \#t = (:\mathrm{quote}.t) \wedge v = t)\end{aligned}$$

Thm 6.5.6.

$$\begin{aligned}&\mathrm{Lproves}(\mathbb{P}, A, \mathrm{FS})\leftrightarrow\\ &\quad(\mathbb{P} = [\,] \wedge A = [\,])\\ &\quad\vee\exists(p, a, \mathbb{P}, A; \#\mathbb{P} = [p.\mathbb{P}] \wedge \#A = [a.A]\\ &\qquad\wedge \mathrm{Proves}(p, a, \mathrm{FS}) \wedge \mathrm{Lproves}(\mathbb{P}, A, \mathrm{FS}))\end{aligned}$$

Thm 6.5.7.

$$\begin{aligned}&\mathrm{Proves}(p, a, \mathrm{FS})\leftrightarrow\\ &\quad\exists(\mathrm{Prd}, \mathbb{R}, i, c, \mathbb{C}, \mathrm{Env}, a, A, \mathbb{P};\\ &\qquad p = [[\mathrm{Prd}, i, \mathrm{Env}].\mathbb{P}] \wedge \#a = [\mathrm{Prd}.a]\\ &\qquad\wedge \mathrm{Assoc}(\mathrm{Prd}, \mathrm{FS}, \mathbb{R}) \wedge \mathrm{Get}(i, \mathbb{R}, [c.\mathbb{C}])\\ &\qquad\wedge \mathrm{Eval}(c, \mathrm{Env}, a)\\ &\qquad\wedge \mathrm{Eval}(\mathbb{C}, \mathrm{Env}, A) \wedge \mathrm{Lproves}(\mathbb{P}, A, \mathrm{FS}))\end{aligned}$$

Thm 6.5.8. $\mathrm{Theorem}(a, \mathrm{FS}) \leftrightarrow \exists(p; \mathrm{Proves}(p, a, \mathrm{FS}))$

All of the above theorems can be proved without much difficulty. As an example let us sketch the proof of Thm 6.5.1.

Proof of Thm 6.5.1. We first prove the $\rightarrow$ part. We prove the following by the $<$-induction on $\mathbb{T}$.

$$\mathrm{Der}(\mathbb{T}, \mathrm{ne}[x, y]) \rightarrow x \neq y$$

So assume $\mathrm{Der}(\mathbb{T}, \mathrm{ne}[x, y])$, that is:

$$\mathrm{Univ\_tree}(\mathbb{T}) \wedge \exists(\mathbb{Q}; \mathbb{T} = [(:\mathrm{ne}[x, y]).\mathbb{Q}])$$

Then applying Thm 6.4.2 we get $\mathfrak{b}$ which is a disjunction of 24 wffs $\mathfrak{b}_i$ $(1 \leqq i \leqq 24)$. Thus we have 24 cases and in (Case i) we may assume $\mathfrak{b}_i$. But for $i > 10$ we can

derive a contradiction by computing the *cbaar* of T. We therefore have only to consider (Case 1) – (Case 10).

(Case 1) Assume $\flat_1$ that is $\exists(u, v; T = [(: ne[*, [u.v]])])$. Let u and v be such that

$$T = [(: ne[*, [u.v]])]$$

Then we have $x = *$ and $y = [u.v]$. By axiom 1, we have $x \neq y$.

(Case 2) – (Case 6) can be proved similarly as (Case 1).

(Case 7) In this case we have:

$$Der(T1, ne[s, u]) \wedge T = [(: ne[[s.t], [u.v]]), T1]$$

for some s, t, v and T1. Then we have $x = [s.t]$ and $y = [u.v]$. Also we have $T1 < T$ and since Der(T1, ne[s, u]) we may apply induction hypothesis to conclude $s \neq u$. Then by axiom 4 we have $[s.t] \neq [u.v]$, that is, $x \neq y$.

(Case 8) – (Case 10) can be proved similarly as (Case 7).

Next we prove the $\leftarrow$ part by induction on y.

(Basis) Assume $x \neq *$. Since Null(x) $\vee$ Mole(x) $\vee$ Atom(x) we have three cases. If Null(x) we have a contradiction. If Mole(x) we have s, t such that $x = [s.t]$. Then we can prove:

$$Der([: ne[[s.t, *]])], ne[[s.t], *])$$

Hence we have Ne([s.t], *), that is, Ne(x, *). The case Atom(x) can be treated similarly.

(Step cons) We prove our target for [y1.y2] assuming it for y1 and y2. Assume $x \neq [y1.y2]$. As before we can classify x into three cases. If $x = *$, we can prove Ne(x, [y1.y2]) similarly as above. If Atom(x) then we have $x = [x1.x2]$ for some x1 and x2. By the decidability of $=$, we have $x1 = y1 \vee x1 \neq y1$. In either case we can prove easily that:

$$[x1.x2] \neq [y1.y2] \rightarrow x1 \neq y1 \vee x2 \neq y2$$

Now, in case $x1 \neq y1$, by induction hypothesis we have Ne(x1, x2). We can then construct a T, for which we have

$$Der(T, ne[[x1.x2], [y1.y2]])$$

From this we have Ne(x, [y1.y2]).

(Step snoc) can be proved similarly. □

Using the predicates Proves, Theorem etc. we can formally talk about formal

systems. Let us give an example. Any closed term in SA denotes a unique sexp, and this relation can be defined by the formal system [*Denote*] where *Denote* is defined as follows:

```
+ denote
    | *, *
    | [s . t], [u . v]
          – denote[s, u]
          – denote[t, v]
    | '(: /s . /t), (: u . v)
          – denote[s, u]
          – denote[t, v]
    | (: quote . t), t
    | (: * . t), (: * . t)
    ;
```

For instance, the closed term 'x denotes the sexp x, since we have:

$$\vdash_{[Denote]} \text{denote['x, x]} \tag{1}$$

by the following reasoning. Let

$$p \equiv \text{[[denote, [*, *, *], [[t . x]]]]}$$

Then after some simple computations we have:

proves(p, denote['x, x], *Denote*)

From this we have (1). We can formalize this by the following definitions:

Def 6.6.1. Denote = '*Denote*

Def 6.6.2. Denote(t, v) = Theorem(denote[t, v], Denote)

Then corresponding to (1), we can prove the following formally in SA.

$$\text{Theorem(denote[''x, 'x], Denote)} \tag{2}$$

A formal proof of (2) can be obtained by translating the informal proof of (1). Namely, in SA we can prove:

Proves('*p*, denote[''x, 'x], Denote)

and from this we have (2).

We can prove a more general theorem similarly. To state the theorem we introduce the following notation for any sexp x:

''x stands for (: quote . x)

Thm 6.6.1. Denote(''x, x)

§7. Metamathematics in SA

In this section we develop elementary metamathematics within SA by formalizing what we have done so far informally.

7.1. universality of FOT

We will prove that a formal version of Theorem 2.1 holds for FOT. In one direction, we have the following:

Thm 7.1.1.

(Der(T, ne[x, y]) → Theorem(ne[x, y], FOT))
∧ (Der(T, assoc[x, L, v]) → Theorem(assoc[x, L, v], FOT))
∧ (Der(T, get[i, L, v]) → Theorem(get[i, L, v], FOT))
∧ (Der(T, eval[t, E, v]) → Theorem(eval[t, E, v], FOT))
∧ (Der(T, lproves[P, A, FS]) → Theorem(lproves[P, A, FS], FOT))
∧ (Der(T, proves[p, a, FS]) → Theorem(proves[p, a, FS], FOT))
∧ (Der(T, theorem[a, FS]) → Theorem(theorem[a, FS], FOT))

This theorem is proved by the <-induction on T. For the other direction, we have the following:

Thm 7.1.2.

(Proves(Q, ne[x, y], FOT) → Ne(x, y))
∧ (Proves(Q, assoc[x, L, v], FOT) → Assoc(x, L, v))
∧ (Proves(Q, get[i, L, v], FOT) → Get(i, L, v))
∧ (Proves(Q, eval[t, E, v], FOT) → Eval(t, E, v))
∧ (Proves(Q, lproves[P, A, FS], FOT) → Lproves(P, A, FS))
∧ (Proves(Q, proves[p, a, FS], FOT) → Proves(p, a, FS))
∧ (Proves(Q, theorem[a, FS], FOT) → Theorem(a, FS))

This theorem can be proved by the <-induction on Q. Combining these theorems we have the following theorems.

Thm 7.1.3. Ne(x, y) ↔ Theorem(ne[x, y], FOT)
Thm 7.1.4. Assoc(x, L, v) ↔ Theorem(assoc[x, L, v], FOT)
Thm 7.1.5. Get(i, L, v) ↔ Theorem(get[i, L, v], FOT)
Thm 7.1.6. Eval(t, E, v) ↔ Theorem(eval[t, E, v], FOT)
Thm 7.1.7. Lproves(P, A, FS) ↔ Theorem(lproves[P, A, FS], FOT)

Thm 7.1.8. Proves(p, a, FS) ↔ Theorem(proves[p, a, FS], FOT)

Thm 7.1.9. Theorem(a, FS) ↔ Theorem(theorem[a, FS], FOT)

7.2. SA in SA

Let us study the axiom system SA within SA. First we give definitions concerning FOT.

Def 7.1.1. Member(x, L) = Theorem(member[x, L], FOT)

Def 7.1.2. Non_member(x, L) = Theorem(non_member[x, L], FOT)

Def 7.1.3. Pure_variable(x) = Theorem(pure_variable[x], FOT)

Def 7.1.4. Pure_variable_list(X) = Theorem(pure_variable_list[X], FOT)

Def 7.1.5. Variable(x) = Theorem(variable[x], FOT)

Def 7.1.6. Pure_part(x, y) = Theorem(pure_part[x, y], FOT)

Def 7.1.7. Length(X, n) = Theorem(length[X, n], FOT)

Def 7.1.8. Symbol(t) = Theorem(symbol[t], FOT)

Def 7.1.9. Term(t, S) = Theorem(term[t, S], FOT)

Def 7.1.10. Term_list(T, S) = Theorem(term_list[T, S], FOT)

Def 7.1.11. Admissible(p) = Theorem(Admissible[p], FOT)

Def 7.1.12. Wff(a, S) = Theorem(wff[a, S], FOT)

Def 7.1.13. Wff_list(A, S) = Theorem(wff_list[A, S], FOT)

Def 7.1.14. Find(t, x, y, v) = Theorem(find[t, x, y, v], FOT)

Def 7.1.15. Rename(x, y, z) = Theorem(rename[x, y, z], FOT)

Def 7.1.16. Free(t, X, v) = Theorem(free[t, X, v], FOT)

Def 7.1.17. Subst(T, X, a, b) = Theorem(subst[T, X, a, b], FOT)

Def 7.1.18. Subst1(t, x, a, b) = Theorem(subst1[t, x, a, b], FOT)

Def 7.1.19. Bind_find(T, X, x, v) = Theorem(bind_find[T, X, x, v], FOT)

Def 7.1.20. Bind_subst(T, X, t, v) = Theorem(bind_subst[T, X, t, v], FOT)

Def 7.1.21. Append(X, Y, Z) = Theorem(append[X, Y, Z], FOT)

Def 7.1.22. Addend(X, Y, Z) = Theorem(addend[X, Y, Z], FOT)

Def 7.1.23. Vars(a, X) = Theorem(vars[a, X], FOT)

Def 7.1.24. Down(X, U, V) = Theorem(down[X, U, V], FOT)

Def 7.1.25. New(x, t) = Theorem(new[x, t], FOT)

Def 7.1.26. New_list(X, t) = Theorem(new_list[X, t], FOT)

Def 7.1.27. Eq_pr(S, E, P, U, V) = Theorem(eq_pr[S, E, P, U, V], FOT)

Def 7.1.28. Lpr(S, E, P, A) = Theorem(lpr[S, E, P, A], FOT)

Def 7.1.29. Upr(S, A, E, P, c) = Theorem(upr[S, A, E, P, c], FOT)

Def 7.1.30. Axiom(a, S) = Theorem(axiom[a, S], FOT)

Def 7.1.31. Sa_axiom(a, S) = Theorem(sa_axiom[a, S], FOT)

Def 7.1.32. Pr(S, E, p, a) = Theorem(pr[S, E, p, a], FOT)

Def 7.1.33. Thm(a, S) = ∃(p; Pr(S, [ ], p, a))

We can prove the following theorems that characterizes these predicates.

Thm 7.2.1.

$$\begin{aligned}&\mathrm{Member}(x, L)\leftrightarrow\\ &\quad \Xi(X; L = [x.X])\\ &\quad \vee \exists(y, X; L = [y.X] \wedge \mathrm{Member}(x, X))\end{aligned}$$

Thm 7.2.2.

$$\begin{aligned}&\mathrm{Non\_member}(x, L)\leftrightarrow\\ &\quad (L = [\,])\\ &\quad \vee \exists(y, X; L = [y.X] \wedge \mathrm{Ne}(x, y) \wedge \mathrm{Non\_member}(x, X))\end{aligned}$$

Thm 7.2.3. $\mathrm{Pure\_variable}(x) \leftrightarrow \exists(t; x = (:\mathrm{var}.t))$

Thm 7.2.4.

$$\begin{aligned}&\mathrm{Pure\_variable\_list}(X)\leftrightarrow\\ &\quad (X = [\,])\\ &\quad \vee \exists(x, X; \#X = [x.X] \wedge \mathrm{Pure\_variable}(x)\\ &\qquad \wedge \mathrm{Non\_member}(x, X) \wedge \mathrm{Pure\_variable\_list}(X))\end{aligned}$$

Thm 7.2.5.

$$\begin{aligned}&\mathrm{Variable}(x) \leftrightarrow\\ &\quad \mathrm{Pure\_variable}(x)\\ &\quad \vee \exists(x; \#x = (:\mathrm{free}.x) \wedge \mathrm{Variable}(x))\end{aligned}$$

Thm 7.2.6.

$$\begin{aligned}&\mathrm{Pure\_part}(x, y) \leftrightarrow\\ &\quad \exists(t; x = (:\mathrm{var}.t) \wedge y = (:\mathrm{var}.t))\\ &\quad \vee \exists(x; \#x = (:\mathrm{free}.x) \wedge \mathrm{Pure\_part}(x, y))\end{aligned}$$

Thm 7.2.7.

$$\begin{aligned}&\mathrm{Length}(X, n) \leftrightarrow\\ &\quad (X = * \wedge n = *)\\ &\quad \vee \exists(X, n; \#X = [x.X] \wedge \#n = [*.n] \wedge \mathrm{Length}(X, n))\end{aligned}$$

Thm 7.2.8. $\mathrm{Symbol}(t) \leftrightarrow \exists(t; \#t = (:*.t))$

Thm 7.2.9.

Term(t, S) ↔
  Variable(t)
  ∨ (t = *)
  ∨ ∃(s, t; #t = [s . t])
  ∨ ∃(s, t; #t = '(: /s . /t))
  ∨ ∃(Fun, T; t = (: apply . [Fun . T]) ∧ Symbol(Fun)
    ∧ Term_list(T, S) ∧ Length(T, Arity)
    ∧ Theorem (function[[Fun . Arity]], S))
  ∨ ∃(t; #t = (: quote . t))
  ∨ ∃(t; #t = (: * . t))

Thm 7.2.10.

Term_list(T, S) ↔
  (T = [ ])
  ∨ ∃(T; #T = [t . T] ∧ Term(t, S) ∧ Term_list(T, S))

Thm 7.2.11.

Admissible(p) ↔
  Symbol(p) ∧ Ne(p, and) ∧ Ne(p, or) ∧ Ne(p, imply)
    ∧ Ne(p, all) ∧ Ne(p, ex)

Thm 7.2.12.

Wff(a, S) ↔
  ∃(s, t; a = eq[s, t] ∧ Term(s, S) ∧ Term(t, S))
  ∨ ∃(s, t; a = lt[s, t] ∧ Term(s, S) ∧ Term(t, S))
  ∨ ∃(Prd, T; a = [Prd . T] ∧ Admissible(Prd)
    ∧ Term_list(T, S) ∧ Length(T, Arity)
    ∧ Theorem(predicate[[Prd . Arity]], S))
  ∨ ∃(A; a = and[. A] ∧ Wff_list(A, S))
  ∨ ∃(A; a = or[. A] ∧ Wff_list(A, S))
  ∨ ∃(A, b; a = imply[A, b] ∧ Wff_list(A, S) ∧ Wff(b, S))
  ∨ ∃(X, a; #a = all[(: abs . [X, a])]
    ∧ Pure_variable_list(X) ∧ Wff(a, S))
  ∨ ∃(X, a; #a = ex[(: abs . [X, a])]
    ∧ Pure_variable_list(X) ∧ Wff(a, S))

Thm 7.2.13.

$$\begin{aligned}&\mathrm{Wff\_list}(A, S) \leftrightarrow\\ &\quad (A = [\,])\\ &\quad \vee \exists(a, A; \#A = [a\,.\,A] \wedge \mathrm{Wff}(a, S) \wedge \mathrm{Wff\_list}(A, S))\end{aligned}$$

Thm 7.2.14.

$$\begin{aligned}&\mathrm{Find}(T, X, x, v) \leftrightarrow\\ &\quad (T = [\,] \wedge X = [\,] \wedge v = x)\\ &\quad \vee \exists(t, T, X; \#T = [t\,.\,T] \wedge \#X = [x\,.\,X] \wedge v = t)\\ &\quad \vee \exists(t, T, y, X; \#T = [t\,.\,T] \wedge \#X = [y\,.\,X]\\ &\qquad \wedge \mathrm{Ne}(x, y) \wedge \mathrm{Find}(T, X, x, v))\end{aligned}$$

Thm 7.2.15.

$$\begin{aligned}&\mathrm{Rename}(x, y, z) \leftrightarrow\\ &\quad (\mathrm{Pure\_variable}(x) \wedge z = x)\\ &\quad \vee\ (\mathrm{Pure\_variable}(y) \wedge x = (:\mathrm{free}\,.\,z))\\ &\quad \vee \exists(x, y, z; \#x = (:\mathrm{free}\,.\,x) \wedge \#y = (:\mathrm{free}\,.\,y)\\ &\qquad \wedge \#z = (:\mathrm{free}\,.\,z) \wedge \mathrm{Rename}(x, y, z))\end{aligned}$$

Thm 7.2.16.

$$\begin{aligned}&\mathrm{Free}(t, X, v) \leftrightarrow\\ &\quad (t = * \wedge v = *)\\ &\quad \vee \exists(x; v = (:\mathrm{free}\,.\,t) \wedge \mathrm{Variable}(t) \wedge \mathrm{Pure\_part}(t, x)\\ &\qquad \wedge \mathrm{Member}(x, X))\\ &\quad \vee \exists(x; v = t \wedge \mathrm{Variable}(t) \wedge \mathrm{Pure\_part}(t, x)\\ &\qquad \wedge \mathrm{Non\_member}(x, X))\\ &\quad \vee \exists(s, t, u, v; \#t = [s\,.\,t] \wedge \#v = [u\,.\,v]\\ &\qquad \wedge \mathrm{Free}(s, X, u) \wedge \mathrm{Free}(t, X, v))\\ &\quad \vee \exists(s, t, u, v; \#t = \text{`}(:/s\,./t) \wedge \#v = \text{`}(:/u\,./v)\\ &\qquad \wedge \mathrm{Free}(s, X, u) \wedge \mathrm{Free}(t, X, v))\\ &\quad \vee \exists(t; \#t = (:*\,.\,t) \wedge v = (:*\,.\,t))\\ &\quad \vee \exists(t; \#t = (:\mathrm{quote}\,.\,t) \wedge v = (:\mathrm{quote}\,.\,t))\\ &\quad \vee \exists(f, T, V; t = (:\mathrm{apply}\,.\,[f\,.\,T])\\ &\qquad \wedge v = (:\mathrm{apply}\,.\,[f\,.\,V]) \wedge \mathrm{Free}(T, X, V))\end{aligned}$$

Thm 7.2.17.

$$\begin{aligned}&\mathrm{Subst}(T, X, a, b) \leftrightarrow\\ &\quad (\mathrm{Variable}(a) \wedge \mathrm{Find}(T, X, a, b))\end{aligned}$$

$$\vee(a = * \wedge b = *)$$
$$\vee\exists(s, t, u, v; a = [s.t] \wedge b = [u.v] \wedge \mathrm{Subst}(T, X, s, u) \wedge \mathrm{Subst}(T, X, t, v))$$
$$\vee\exists(s, t, u, v; a = `(: /s./t) \wedge b = `(: /u./v) \wedge \mathrm{Subst}(T, X, s, u) \wedge \mathrm{Subst}(T, X, t, v))$$
$$\vee\exists(t; a = (: *.t) \wedge b = (: *.t))$$
$$\vee\exists(t; a = (: \mathrm{quote}.t) \wedge b = (: \mathrm{quote}.t))$$
$$\vee\exists(f, U, V; a = (: \mathrm{apply}.[f.U]) \wedge b = (: \mathrm{apply}.[f.V]) \wedge \mathrm{Subst}(T, X, U, V))$$
$$\vee\exists(Y, t, v; a = (: \mathrm{abs}.[Y, t]) \wedge b = (: \mathrm{abs}.[Y, v]) \wedge \mathrm{Pure\_variable\_list}(Y) \wedge \mathrm{Free}(T, Y, V) \wedge \mathrm{Free}(X, Y, Z) \wedge \mathrm{Subst}(V, Z, t, v))$$

Thm 7.2.18. $\mathrm{Subst1}(t, x, s, v) \leftrightarrow \mathrm{Subst}([t], [x], s, v)$

Thm 7.2.19.

$$\mathrm{Bind\_find}(T, X, x, v) \leftrightarrow$$
$$(T = [\,] \wedge X = [\,] \wedge x = v)$$
$$\vee\exists(T, X; \#T = [v.T] \wedge \#X = [x.X])$$
$$\vee\exists(t, T, y, u; \#T = [t.T] \wedge \#X = [y.X] \wedge x \neq y \wedge \mathrm{Pure\_part}(x, u) \wedge \mathrm{Pure\_part}(y, u) \wedge \mathrm{Rename}(x, y, v))$$
$$\vee\exists(t, T, y, u1, u2; \#T = [t.T] \wedge \#X = [y.X] \wedge x \neq y \wedge \mathrm{Pure\_part}(x, u1) \wedge \mathrm{Pure\_part}(y, u2) \wedge u1 \neq u2 \wedge \mathrm{Bind\_find}(T, X, x, v))$$

Thm 7.2.20.

$$\mathrm{Bind\_subst}(T, X, t, v) \leftrightarrow$$
$$(\mathrm{Variable}(t) \wedge \mathrm{Bind\_find}(T, X, t, v))$$
$$\vee(t = * \wedge v = *)$$
$$\vee\exists(s, t, u, v; \#t = [s.t] \wedge \#v = [u.v] \wedge \mathrm{Bind\_subst}(T, X, s, u) \wedge \mathrm{Bind\_subst}(T, X, t, v))$$
$$\vee\exists(s, t, u, v; \#t = (: s.t) \wedge \#v = (: u.v) \wedge \mathrm{Bind\_subst}(T, X, s, u) \wedge \mathrm{Bind\_subst}(T, X, t, v))$$
$$\vee(t = v \wedge \mathrm{Symbol}(t))$$
$$\vee\exists(t; \#t = (: \mathrm{quote}.t) \wedge \#v = (: \mathrm{quote}.t))$$
$$\vee\exists(f, U, V; t = (: \mathrm{apply}.[f.U]) \wedge v = (: \mathrm{apply}.[f.V]) \wedge \mathrm{Bind\_subst}(T, X, U, V))$$
$$\vee\exists(Y, t, v, V, Z; \#t = (: \mathrm{abs}.[Y, t]) \wedge \#v = (: \mathrm{abs}.[Y, v])$$

$\wedge$ Pure_variable_list(Y) $\wedge$ Free(T, Y, V)
$\wedge$ Free(X, Y, Z) $\wedge$ Bind_subst(V, Z, t, v))

Thm 7.2.21.

Append(X, Y, Z) $\leftrightarrow$
(X = [ ] $\wedge$ Z = Y)
$\vee\exists$(x, X, Z; #X = [x.X] $\wedge$ #Z = [x.Z] $\wedge$ Append(X, Y, Z))

Thm 7.2.22.

Addend(X, Y, Z) $\leftrightarrow$
(X = [ ] $\wedge$ Z = Y)
$\vee\exists$(x, X; #X = [x.X] $\wedge$ Addend(X, Y, Z) $\wedge$ Member(x, Z))
$\vee\exists$(x, X Z; #X = [x.X] $\wedge$ #Z = [x.Z] $\wedge$ Addend(X, Y, Z))

Thm 7.2.23.

Vars(t, V) $\leftrightarrow$
(Variable(t) $\wedge$ V = [t])
$\vee$(t = * $\wedge$ V = [ ])
$\vee\exists$(s, t, S, T; #t = [s.t] $\wedge$ Vars(s, S) $\wedge$ Vars(t, T)
$\wedge$ Addend(S, T, V))
$\vee\exists$(s, t, S, T; #t = '(: /s./t) $\wedge$ Vars(s, S) $\wedge$ Vars(t, T)
$\wedge$ Addend(S, T, V))
$\vee\exists$(t; #t = (: *.t) $\wedge$ V = [ ])
$\vee\exists$(t; #t = (: quote.t) $\wedge$ V = [ ])
$\vee\exists$(f, T; t = (: apply.[f.T]) $\wedge$ Vars(T, V))
$\vee\exists$(X, a, U; t = (: abs.[X, a]) $\wedge$ Vars(a, U)
$\wedge$ Down(X, U, V))

Thm 7.2.24.

Down(X, U, V) $\leftrightarrow$
(U = [ ] $\wedge$ V = [ ])
$\vee\exists$(x, U; #U = [x.U] $\wedge$ Pure_variable(x)
$\wedge$ Member(x, X) $\wedge$ Down(X, U, V))
$\vee\exists$(x, U, V ; #U = [x.U] $\wedge$ #V = [x.V]
$\wedge$ Pure_variable(x) $\wedge$ Non_member(x, X)
$\wedge$ Down(X, U, V))
$\vee\exists$(y, U, V ; #U = [(: free.y).U] $\wedge$ #V = [y.V]
$\wedge$ Pure_part(y, z) $\wedge$ Member(z, X) $\wedge$ Down(X, U, V))

$$\vee\,\exists(y, U, V;\ \#U = [(:\text{free}.y).U] \wedge \#V = [(:\text{free}.y).V]$$
$$\wedge\ \text{Pure\_part}(y, z) \wedge \text{Non\_member}(z, X)$$
$$\wedge\ \text{Down}(X, U, V))$$

Thm 7.2.25. $\text{New}(x, t) \leftrightarrow \exists(X;\ \text{Vars}(t, X) \wedge \text{Non\_member}(x, X))$

Thm 7.2.26.

$$\text{New\_list}(X, t) \leftrightarrow$$
$$(X = [\ ])$$
$$\vee\,\exists(x, X;\ \#X = [x.X] \wedge \text{New}(x, t) \wedge \text{New\_list}(X, t))$$

Thm 7.2.27.

$$\text{Eq\_pr}(S, E, P, U, V) \leftrightarrow$$
$$(P = [\ ] \wedge U = [\ ] \wedge V = [\ ])$$
$$\vee\,\exists(p, P, u, U, v, V;\ \#P = [p.P] \wedge \#U = [u.U]$$
$$\wedge\ \#V = [v.V] \wedge \text{Pr}(S, E, p, u = v)$$
$$\wedge\ \text{Eq\_pr}(S, E, P, U, V))$$

Thm 7.2.28.

$$\text{Lpr}(S, E, P, A) \leftrightarrow$$
$$(P = [\ ] \wedge A = [\ ])$$
$$\vee\,\exists(p, P, a, A;\ \#P = [p.P] \wedge \#A = [a.A]$$
$$\wedge\ \text{Pr}(S, E, p, a) \wedge \text{Lpr}(S, E, P, A))$$

Thm 7.2.29.

$$\text{Upr}(S, A, E, P, c) \leftrightarrow$$
$$(A = [\ ] \wedge P = [\ ])$$
$$\vee\,\exists(a, A, p, P;\ \#A = [a.A] \wedge \#P = [p.P]$$
$$\wedge\ \text{Pr}(S, [a.E], p, c) \wedge \text{Upr}(S, A, E, P, c))$$

Thm 7.2.30.

$$\text{Axiom}(a, S) \leftrightarrow$$
$$\text{Sa\_axiom}(a, S)$$
$$\vee\,\exists(p;\ \text{Proves}(p, \text{axiom}[a], S) \wedge \text{Wff}(a, S)$$

Thm 7.2.31.

$$\text{SA\_axiom}(a, S) \leftrightarrow$$
$$(a = '\forall(s, t;\ [s.t] \neq *))$$
$$\vee\ (a = '\forall(s, t;\ (:s.t) \neq *))$$
$$\vee\ (a = '\forall(s, t, u, v;\ [s.t] \neq (:u.v)))$$
$$\vee\ (a = '\forall(s, t, u, v;\ [s.t] = [u.v] \rightarrow s = u))$$

$$
\begin{aligned}
&\vee\ (a = '\forall(s, t, u, v; [s.t] = [u.v] \rightarrow t = v))\\
&\vee\ (a = '\forall(s, t, u, v; (:s.t) = (:u.v) \rightarrow s = u))\\
&\vee\ (a = '\forall(s, t, u, v; (:s.t) = (:u.v) \rightarrow t = v))\\
&\vee\ (a = '\forall(r; \neg(r < *)))\\
&\vee\ (a = '\forall(r, s, t; \neg(r < (:s.t))))\\
&\vee\ (a = '\forall(s, t; s < [s.t]))\\
&\vee\ (a = '\forall(s, t; t < [s.t]))\\
&\vee\ (a = '\forall(r, s, t; r < s \rightarrow r < [s.t]))\\
&\vee\ (a = '\forall(r, s, t; r < t \rightarrow r < [s.t]))\\
&\vee\ (a = '\forall(r, s, t; r < [s.t] \rightarrow r = s \vee r < s \vee r = t \vee r < t))\\
&\vee (a = '((\text{quote}.*) = *))\\
&\vee \exists(s, t; a = `((\text{quote}.[/s./t]) = [(\text{quote}./s).(\text{quote}./t)]))\\
&\vee \exists(s, t; a = `((\text{quote}.(/s./t)) = (:(\text{quote}./s).(\text{quote}./t))))\\
&\vee\ (a = '((*.*) = (:*.*)))\\
&\vee \exists(s, t, u, v; a = `((*./s) = (:*./u), (*./t) = (:*./v) \rightarrow\\
&\qquad (*.[/s./t]) = (:*.[/u./v])))\\
&\vee \exists(s, t, u, v; a = `((*./s) = (:*./u), (*./t) = (:*./v) \rightarrow\\
&\qquad (*.(/s./t)) = (:*.(:/u./v))))\\
&\vee \exists(a0, x, y, IHx, IHy, a1, a2, b, a;\\
&\qquad \#a = `(/a0, \forall(/x, /y; /IHx, /IHy \rightarrow /a1),\\
&\qquad\qquad \forall(/x, /y; /IHx, /IHy \rightarrow /a2) \rightarrow /b)\\
&\qquad \wedge\ \text{Pure\_variable}(x) \wedge \text{Pure\_variable}(y)\\
&\qquad \wedge\ \text{Pure\_variable}(z)\\
&\qquad \wedge\ \text{Ne}(x, y) \wedge \text{New}(x, a) \wedge \text{New}(y, a) \wedge \text{Wff}(a, S)\\
&\qquad \wedge\ \text{Term}(t, S)\\
&\qquad \wedge\ \text{Subst1}(t, z, a, b) \wedge \text{Subst1}(*, z, a, IHx)\\
&\qquad \wedge\ \text{Subst1}(y, z, a, IHy)\\
&\qquad \wedge\ \text{Subst1}([x.y], z, a, a1) \wedge \text{Subst1}(`(:/x./y), z, a, a2))\\
&\vee \exists(z, w, a1, a, b;\\
&\qquad \#a = `(\forall(/z; \forall(/w; /w < /z \rightarrow /a1) \rightarrow /a) \rightarrow /b)\\
&\qquad \wedge\ \text{Pure\_variable}(z) \wedge \text{Pure\_variable}(w)\\
&\qquad \wedge\ \text{Ne}(z, w) \wedge \text{New}(w, a)\\
&\qquad \wedge\ \text{Wff}(a, S) \wedge \text{Term}(t, S) \wedge \text{Subst1}(w, z, a, a1)\\
&\qquad \wedge\ \text{Subst1}(t, z, a, b))
\end{aligned}
$$

Thm 7.2.32.

$$
\begin{aligned}
&\mathrm{Pr}(S, E, p, a) \leftrightarrow \\
&\quad (p = [a, \mathrm{axiom}[\,]] \wedge \mathrm{Axiom}(a, S) \wedge \mathrm{Wff\_list}(E, S)) \\
&\quad \vee (p = [a, \mathrm{assumption}[\,]] \wedge \mathrm{Member}(a, E) \wedge \mathrm{Wff\_list}(E, S)) \\
&\quad \vee \exists(A, P;\ p = [\mathrm{and}[.A], \mathrm{and\_I}[.P]] \wedge a = \mathrm{and}[.A] \\
&\qquad \wedge \mathrm{Lpr}(S, E, P, A)) \\
&\quad \vee \exists(p, A;\ \sharp p = [a, \mathrm{and\_E}[p]] \wedge \mathrm{Pr}(S, E, p, \mathrm{and}[.A]) \\
&\qquad \wedge \mathrm{Member}(a, A)) \\
&\quad \vee \exists(A, p;\ \sharp p = [\mathrm{or}[.A], \mathrm{or\_I}[p]] \wedge a = \mathrm{or}[.A] \\
&\qquad \wedge \mathrm{Pr}(S, E, p, a) \wedge \mathrm{Member}(a, A) \wedge \mathrm{Wff\_list}(A, S)) \\
&\quad \vee \exists(p, P, A;\ \sharp p = [a, \mathrm{or\_E}[p.P]] \wedge \mathrm{Pr}(S, E, p, \mathrm{or}[.A]) \\
&\qquad \wedge \mathrm{Upr}(S, A, E, P, a)) \\
&\quad \vee \exists(A, b, p, F;\ \sharp p = [\mathrm{imply}[A, b], \mathrm{imply\_I}[p]] \\
&\qquad \wedge a = \mathrm{imply}[A, b] \wedge \mathrm{Append}(A, E, F) \wedge \mathrm{Pr}(S, F, p, b)) \\
&\quad \vee \exists(p, P, A;\ \sharp p = [a, \mathrm{imply\_E}[p, P]] \\
&\qquad \wedge \mathrm{Pr}(S, E, p, \mathrm{imply}[A, a]) \wedge \mathrm{Lpr}(S, E, P, A)) \\
&\quad \vee \exists(X, a, Y, p, b;\ \sharp p = [\mathrm{all}[(:\mathrm{abs}.[X, a])], \mathrm{all\_I}[[X, Y], p]] \\
&\qquad \wedge \sharp a = \mathrm{all}[(:\mathrm{abs}.[X, a])] \\
&\qquad \wedge \mathrm{New\_list}(Y, E) \wedge \mathrm{Pure\_variable\_list}(X) \wedge \mathrm{Wff}(a, S) \\
&\qquad \wedge \mathrm{Subst}(Y, X, a, b) \wedge \mathrm{Pr}(S, E, p, b)) \\
&\quad \vee \exists(T, p, X, b;\ \sharp p = [a, \mathrm{all\_E}[T, p]] \\
&\qquad \wedge \mathrm{Pr}(S, E, p, \mathrm{all}[(:\mathrm{abs}.[X, b])]) \\
&\qquad \wedge \mathrm{Term\_list}(T, S) \wedge \mathrm{Subst}(T, X, b, a)) \\
&\quad \vee \exists(X, b, T, p;\ \sharp p = [\mathrm{ex}[(:\mathrm{abs}.[X, b])], \mathrm{ex\_I}[T, p]] \\
&\qquad \wedge a = \mathrm{ex}[(:\mathrm{abs}.[X, b])] \wedge \mathrm{Term\_list}(T, S) \\
&\qquad \wedge \mathrm{Pure\_variable\_list}(X) \wedge \mathrm{Wff}(b, S) \\
&\qquad \wedge \mathrm{Subst}(T, X, b, a) \wedge \mathrm{Pr}(S, E, p, a)) \\
&\quad \vee \exists(Y, p, q, b, c;\ \sharp p = [a, \mathrm{ex\_E}[Y, p, q]] \wedge \mathrm{New\_list}[Y, E] \\
&\qquad \wedge \mathrm{New\_list}(Y, a) \wedge \mathrm{Pr}(S, E, p, \mathrm{ex}[(:\mathrm{abs}.[X, b])]) \\
&\qquad \wedge \mathrm{Subst}(Y, X, b, c) \wedge \mathrm{Pr}(S, [c.E], q, a)) \\
&\quad \vee \exists(t;\ p = [t = t, \mathrm{axiom\_id}[\,]] \wedge \mathrm{Term}(t, S)) \\
&\quad \vee \exists(U, V, X, c, p, P, b;\ \sharp p = [a, \mathrm{subst}[[U, V, X, c], p.P]] \\
&\qquad \wedge \mathrm{Wff}(c, S) \wedge \mathrm{Subst}(U, X, c, b) \wedge \mathrm{Subst}(V, X, c, a) \\
&\qquad \wedge \mathrm{Pr}(S, E, p, b) \wedge \mathrm{Eq\_pr}(S, E, P, U, V))
\end{aligned}
$$

Using these theorems we can study about SA and its extensions within

$\mathbb{SA}$. For example corresponding to the simple metatheorem:

$$S \vdash a \to b,\ S \vdash a \Longrightarrow S \vdash b$$

we have the following theorem:

Thm 7.3.1. Thm(a → b, S), Thm(a, S) → Thm(b, S)

Proof. Assume Thm(a → b, S) and Thm(a, S). Then we have p and q such that:

$$\mathrm{Pr}(S, [\ ], p, a \to b) \tag{1}$$

$$\mathrm{Pr}(S, [\ ], q, a) \tag{2}$$

From (2) and Thm 7.2.28 we have:

$$\mathrm{Lpr}(S, [\ ], [q], [a]) \tag{3}$$

By (1), (3) and Thm 7.2.32 we have:

$$\mathrm{Pr}(S, [\ ], [b, \mathrm{imply\_E}[p, [q]]], b) \tag{4}$$

Then by applying ($\exists I$) to (4) we have Thm(b, S). □

We can similarly prove the following theorems and much more similar theorems which we do not list here.

Thm 7.3.2. Thm(a, SA) → Thm(a, S)

Thm 7.3.3. Wff(a, S) → Thm(⊥ → a, S)

Thm 7.3.4. Thm(''t = ''t, SA)

Thm 7.3.5. Thm(''[s . t] = ['' s . ''t], SA)

Thm 7.3.6. Thm('(/''(: s . t) = (: /''s . /''t)), SA)

Next we will prove that $\mathbb{SA}$ is *inductively complete* in the sense that if Theorem (a, FS) then we can formally prove it in $\mathbb{SA}$. (We borrowed this terminology from Feferman [4].) We can state this formally as follows:

Theorem(a, FS) → Thm('Theorem(/''a, /''FS), SA)

We first prove the following key lemma.

Thm 7.3.7. Univ_tree(T) → Thm('Univ_tree(/''T), SA)

Proof. We prove this theorem by the <-induction on T. Assume Univ_tree(T). Then by Thm 6.4.2 we have 24 cases to consider of which we treat only (Case 17) as a typical case.

(Case 17) In this case we have some s, t, E, u, v, T1 and T2 such that:

$$\mathrm{Der}(T1, \mathrm{eval}[s, E, u]) \tag{1}$$
$$\mathrm{Der}(T2, \mathrm{eval}[t, E, v]) \tag{2}$$

and

$$T = [(: \mathrm{eval}[s\,.\,t], E, [u\,.\,v]), T1, T2] \tag{3}$$

From (1) we have:

$$\mathrm{Univ\_tree}(T) \tag{4}$$

and

$$\exists(Q;\ T1 = [(: \mathrm{eval}[s, E, u])\,.\,Q]) \tag{5}$$

By (3) we have $T1 < T$ and we can apply induction hypothesis to (4) and get:

$$\mathrm{Thm}(\text{`}\mathrm{Univ\_tree}(/\text{``}T1), SA) \tag{6}$$

Now by (5) we can take Q such that:

$$T1 = [(: \mathrm{eval}[s, E, u])\,.\,Q] \tag{7}$$

By Thm 7.3.4 we have:

$$\mathrm{Thm}(\text{``}T1 = \text{``}T1, SA) \tag{8}$$

We can rewrite (8) using (7), Thm 7.3.5 and Thm 7.3.6 as follows:

$$\mathrm{Thm}(\text{`}(/\text{``}T1 = [(: \mathrm{eval}[/\text{``}s, /\text{``}E, /\text{``}u])\,./\text{``}Q]), SA) \tag{9}$$

From (6) and (9) using Thm 7.2.32 we get:

$$\mathrm{Thm}(\text{`}\mathrm{Der}(/\text{``}T1, \mathrm{eval}[/\text{``}s, /\text{``}E, /\text{``}u]), SA) \tag{10}$$

From (2) we have the following similarly:

$$\mathrm{Thm}(\text{`}\mathrm{Der}(/\text{``}T2, \mathrm{eval}[/\text{``}t, /\text{``}E, /\text{``}v]), SA) \tag{11}$$

We can also prove the following by using (3):

$$\mathrm{Thm}(\text{`}(/\text{``}T = [(: \mathrm{eval}[/\text{``}s\,./\text{``}t], /\text{``}E, [/\text{``}u\,./\text{``}v]), /\text{``}T1, /\text{``}T2]), SA) \tag{12}$$

From (10), (11) and (12) we can prove:

$$\begin{aligned}&\mathrm{Thm}(\text{`}\exists(s, t, E, u, v, T1, T2;\\ &\mathrm{Der}(T1, \mathrm{eval}[s, E, u]) \wedge \mathrm{Der}(T2, \mathrm{Eval}[t, E, v])\\ &\wedge\ T = [(: \mathrm{eval}[[s\,.\,t], E, [u\,.\,v]]), T1, T2]), SA)\end{aligned} \tag{13}$$

Now let $\mathfrak{b}$ be the right hand side of the equivalence of Thm 6.4.2. Then applying $(\vee I)$ to (13) we have:

$$\mathrm{Thm}(`\dot{b}_T[/``T], \mathrm{SA}) \tag{14}$$

On the other hand, repeating the proof of Thm 6.4.2 within SA, we obtain:

$$\mathrm{Thm}(`(\mathrm{Univ\_tree}(/``T) \leftrightarrow \dot{b}_T(/``T]), \mathrm{SA}) \tag{15}$$

By (14) and (15) we have:

$$\mathrm{Thm}(`(\mathrm{Univ\_tree}(/``T)), \mathrm{SA})$$

□

Using Thm 7.3.7 we can easily prove the inductive completeness of SA:

Thm 7.3.8. Theorem(a, FS) → Thm('Theorem(/''a, /''FS), SA)

We also have the following theorem as a corollary to this theorem.

Thm 7.3.9. Thm(a, S) → Thm('Thm(/''a, /''S), SA)

§8. Incompletness Theorem

In this section we prove some of Gödel's incompleteness theorems (Gödel[5]) including the second incompleteness theorem formally in SA.

8.1. reflection principle

Let us make some observations about what we are doing by looking at it from *outside*. We have been developing our informal theory of symbolic expressions only using constructively acceptable arguments. We also claim that our formal theory SA reflects faithfully part of our informal mathematics. This means firstly that each wff in SA can be translated into an informal statement that is meaningful to our informal mathematics, and secondly that each formal proof of a wff in SA can be translated into an acceptable informal proof of the corresponding informal statement. We may call this translation process as *informalization*. Since the translation process should be almost clear, we only explain how terms in SA are translated into informal expressions denoting sexps. We denote the translation of a term t by $\tilde{t}$. If t is a variable then we translate it into an informal variable. E.g., if t is 'x' then we translate it into 'x'. We translate $*$ into $*$. The translation of $[s\,.\,t]$ is $[\tilde{s}\,.\,\tilde{t}]$ and the translation of $(:\,s\,.\,t)$ is $(\tilde{s}\,.\,\tilde{t})$. The translation of $'t$ is t and the translation of $(*\,.\,t)$ is $(*\,.\,\tilde{t})$.

Let us consider the translation of formal developments in Section 6 and 7. The corresponding informal developments considerably overlaps with our earlier developments in Sections 1–5. But there are minor differences which we now

explain. In Section 2 we defined the concept of a formal system using informal inductive definitions. Namely, we defined the informal predicates *ne*, *assoc*, *get*, *eval*, *lproves*, *proves* and *theorem* by informal inductive definitions. On the other hand, by translating Def 6.1.1–7 in Section 6.4 we have the explicit definitions of the informal predicates *Ne*, *Assoc*, *Get*, *Eval*, *Lproves*, *Proves* and *Theorem*. It is, however, easy to see informally that these two groups of concepts are equivalent. From this observation, it follows, for instance, that the concept defined by the translation of Def 7.1.32 in Section 7.2 is equivalent to the notion of the provability in a formal theory which we defined in Section 4.2.

We can thus conclude that each formal theorem of SA yields as by-product an informal theorem which is its informal counterpart. Such an informal theorem may sometimes be used to produce another formal theorem. We give two important examples of this.

By informalizing Thm 7.3.8 we have the following theorem:

Theorem 8.1. $\vdash_{FS} a \Rightarrow \mathrm{SA} \vdash \mathrm{Theorem}('a, 'FS)$

The logic programming language Qute [12] can be used to verify $\vdash_{FS} a$ automatically, so that this theorem will be useful when we implement a proof checking system for SA on a computer.

The following theorem can be obtained by reading Thm 7.3.9 informally.

Theorem 8.2. $S \vdash a \Rightarrow \mathrm{SA} \vdash \mathrm{Thm}('a, 'S)$

This theorem is the first Löb derivability condition and Thm 7.3.9 is the second. We have already proved the third derivability condition as Thm 7.3.1. (See, e.g., Feferman [4] for these derivability conditions.)

8.2. diagonalization lemma

We prove the diagonalization lemma in this subsection. To state the diagonalization lemma, we need some definitions.

Def 8.1.1. $\mathrm{Sub}(t, a, b) = \exists(x; \mathrm{Vars}(a, [x]) \wedge \mathrm{Subst}1(\text{``}t, x, a, b))$

Def 8.1.2. $\mathrm{A}(x, S) = \exists(X; \mathrm{Sub}(x, x, X) \wedge \neg\mathrm{Thm}(X, S))$

Def 8.1.3. $\mathrm{B}(S) = \text{`}\mathrm{A}(x, /\text{``}S)$

Def 8.1.4. $\mathrm{C}(S) = \mathrm{A}(\mathrm{B}(S), S)$

Def 8.1.5. $\mathrm{D}(S) = \text{`}\mathrm{A}(/\text{``}\mathrm{B}(S), /\text{``}S)$

Using these predicates, we can state the diagonalization lemma as follows.

$$\mathrm{Thm}(\mathbb{D}(S)) \leftrightarrow \neg\,\mathrm{`Thm}(/\!\!`\mathbb{D}(S), /\!\!`S), SA)$$

To prove this we prepare some auxiliary lemmas. We can easily prove the following two lemmas:

Thm 8.1.1. $\mathrm{Sub}(t, a, b1), \mathrm{Sub}(t, a, b2) \rightarrow b1 = b2$

Thm 8.1.2. $\mathrm{Sub}(B(S), B(S), \mathbb{D}(S))$

Then we can prove the following lemma.

Thm 8.1.3. $C(S) \leftrightarrow \neg\,\mathrm{Thm}(\mathbb{D}(S), S)$

Proof. We first prove the $\rightarrow$ part. Assume $C(S)$ and let X be such that:

$$\mathrm{Sub}(B(S), B(S), X) \wedge \neg\,\mathrm{Thm}(X, S)$$

This implies:

$$\mathrm{Sub}(B(S), B(S), X) \tag{1}$$

and

$$\neg\,\mathrm{Thm}(X, S) \tag{2}$$

Then by (1), Thm 8.1.2 and Thm 8.1.1 we have:

$$X = \mathbb{D}(S) \tag{3}$$

By (2) and (3) we have $\neg\,\mathrm{Thm}(\mathbb{D}(S), S)$.

Next, we prove the $\leftarrow$ part. Assume $\mathrm{Thm}(\mathbb{D}(S), S)$. Then by Thm 8.1.2 we have:

$$\mathrm{Sub}(B(S), B(S), \mathbb{D}(S)) \wedge \neg\,\mathrm{Thm}(\mathbb{D}(S), S)$$

From this we have $C(S)$ by $(\exists I)$. □

We have the following by applying $(\forall I)$ to Thm 8.1.3.

$$\mathbb{SA} \vdash \forall(S; C(S) \leftrightarrow \neg\,\mathrm{Thm}(\mathbb{D}(S), S))$$

From this by Theorem 8.2 we have the following:

$$\mathbb{SA} \vdash \mathrm{Thm}(\text{`}\forall(S; C(S) \leftrightarrow \neg\,\mathrm{Thm}(\mathbb{D}(S), S)), SA)$$

We therefore have the following theorem.

Thm 8.1.4. $\mathrm{Thm}(\text{`}\forall(S; C(S) \leftrightarrow \neg\,\mathrm{Thm}(\mathbb{D}(S), S)), SA)$

We can now prove the diagonalization lemma as follows.

Thm 8.1.5. $\mathrm{Thm}(\mathbb{D}(S)) \leftrightarrow \neg\,\mathrm{`Thm}(/\!\!`\mathbb{D}(S), /\!\!`S), SA)$

Proof. By a simple but tedious computation we have:

$$'\forall(S;\ C(S) \leftrightarrow \neg\,\mathrm{Thm}(\mathbb{D}(S),\ S)) = \ulcorner\forall(S;\ C(S) \leftrightarrow \neg\,\mathrm{Thm}(\mathbb{D}(S),\ S)) \quad (1)$$

By (1) and Thm 8.1.4 we have:

$$\mathrm{Thm}(\ulcorner\forall(S;\ C(S) \leftrightarrow \neg\,\mathrm{Thm}(\mathbb{D}(S),\ S)),\ SA) \quad (2)$$

By formally specializing 'S' to the term ''S we obtain:

$$\mathrm{Thm}(\ulcorner(C(/''S) \leftrightarrow \neg\,\mathrm{Thm}(\mathbb{D}(/''S),\ /''S)),\ SA) \quad (3)$$

On the other hand we have the following by a simple calculation:

$$\mathrm{Thm}(\ulcorner\mathbb{D}(/''S) = ''\mathbb{D}(S),\ SA) \quad (4)$$

By (3) and (4) we have the following as desired:

$$\mathrm{Thm}(\mathbb{D}(S) \leftrightarrow \neg\,\ulcorner\mathrm{Thm}(/''\mathbb{D}(S),\ /''S),\ SA)$$

□

Remark. The method we used to obtain Thm 8.1.5 from Thm 8.1.3 is general and applicable to similar cases. We will call this method as *formalization.* Thus, e.g., we say that Thm 8.1.5 is obtained by formalizing Thm 8.1.3. □

8.3. incompleteness theorems

We first define the concept of *consistency.*

Def 8.2.1. $\mathrm{Consis}(S) = \neg\,\mathrm{Thm}(\bot,\ S)$

Consis(S) says that S is consistent.

We will prove the following theorems.

Thm 8.2.1. $\mathrm{Thm}(\mathbb{D}(S),\ S) \rightarrow \mathrm{Thm}(\bot,\ S)$

Thm 8.2.2. $\mathrm{Consis}(S) \rightarrow \neg\,\mathrm{Thm}(\mathbb{D}(S),\ S)$

Thm 8.2.3. $\mathrm{Consis}(S) \rightarrow C(S)$

Thm 8.2.4. $\mathrm{Thm}(\ulcorner\mathrm{Consis}(/''S) \rightarrow \mathbb{D}(S),\ SA)$

Thm 8.2.5. $\mathrm{Thm}(\ulcorner\mathrm{Consis}(/''S) \rightarrow \mathbb{D}(S),\ S)$

Thm 8.2.6. $\mathrm{Thm}(\ulcorner\mathrm{Consis}(/''S),\ S) \rightarrow \mathrm{Thm}(\mathbb{D}(S),\ S)$

Thm 8.2.7. $\mathrm{Consis}(S) \rightarrow \neg\,\mathrm{Thm}(\ulcorner\mathrm{Consis}(/''S),\ S)$

Proof of Thm 8.2.1. Assume:

$$\mathrm{Thm}(\mathbb{D}(S),\ S) \quad (1)$$

By Thm 7.3.9 and (1) we have:

$$\mathrm{Thm}(\text{`}\mathrm{Thm}(/\text{``}D(S), /\text{``}S), SA) \tag{2}$$

By (2) and Thm 7.3.2 we have:

$$\mathrm{Thm}(\text{`}\mathrm{Thm}(/\text{``}D(S), /\text{``}S), S) \tag{3}$$

On the other hand by Thm 8.1.5 and Thm 7.3.2 we have:

$$\mathrm{Thm}(D(S) \to \neg\text{`}\mathrm{Thm}(/\text{``}D(S), /\text{``}S), S) \tag{4}$$

By (1), (4) and Thm 7.3.1 we have:

$$\mathrm{Thm}(\neg\text{`}\mathrm{Thm}(/\text{``}D(S), /\text{``}S), S) \tag{5}$$

By (3), (5) and Thm 7.3.1 we have $\mathrm{Thm}(\bot, S)$. □

Thm 8.2.2 is a logical consequence of Thm 8.2.1. Thm 8.2.2 says that if S is consistent then the 'formula' D(S) which states its own unprovability is in fact unprovable. Here we note that S must be an extension of SA according to our definition of FOT, and that Thm 7.3.2 states this fact formally.

Thm 8.2.3 follows from Thm 8.2.2 and Thm 8.1.3.

We can obtain Thm 8.2.4 by formalizing Thm 8.2.3. (Recall the remark we made after the proof of Thm 8.1.5.) Thm 8.2.5 then follows from this theorem by applying Thm 7.3.2.

Thm 8.2.6 is a logical consequence of Thm 8.2.5 and Thm 7.3.1.

Finally, we get Thm 8.2.7 as a logical consequence of Thm 8.2.2 and Thm 8.2.6.

Thm 8.2.7 is the formalized second incompleteness theorem, and we can obtain the informal second incompleteness theorem by informalizing Thm 8.2.9. We therefore have the following two metatheorems.

Theorem 8.3. (Second Incompleteness Theorem)

If S is consistent then $S \not\vdash \mathrm{Consis}(\text{`}S)$

Theorem 8.4. (Formalized Second Incompleteness Theorem)

$$SA \vdash \mathrm{Consis}(S) \to \neg\mathrm{Thm}(\text{`}\mathrm{Consis}(/\text{``}S), S)$$

8.4. concluding remarks

The purpose of the present paper was to provide a formal axiomatic theory in which one can actually work without resorting to metamathematical arguments. We have set the task of proving Gödel's (formalized) second incompleteness theorem to test the adequacy of the theory in this respect.

It is obvious that the existing theories like PA (Peano arithmetic) or HA (Heyting arithmetic) are good for studying them but are not good for actually working within them (especially when one has to prove metamathematical theorems like incompleteness theorems). To work within them is as difficult as programming in Turing machines. (We note that Beeson[1] has made a similar remark.)

Since the basic entities one studies in metamathematics are syntactic objects like wffs or proof, and since one must develop some metamathematics within a theory to prove incompleteness theorems, it is desirable that such a theory can handle syntactic objects naturally. It has long been known in computer science that pairing structures provide a natural framework for representing these syntactic objects as a tree structure. These pairing structures are known as McCarthy's symbolic expressions, and are basic objects of the programming language LISP (McCarthy[8]). Feferman[4] noticed the usefulness of McCarthy's symbolic expressions and developed two formal theories of symbolic expressions, FM and FM_0, based on second order classical logic. Despite the differences in the logics used and the differences in the basic objects (Feferman uses McCarthy's sexps and we use our sexps), the present work and Feferman's seem to have succeeded in providing workable formal theories in a fairly similar manner. We think that the success owes very much to the mathematical elegance of symbolic expressions.

Another important reason for our choice of sexps as the basic objects in our formal theory SA is that they are implementable on a computer. This is essential because this makes it possible to constuct a proof checking system for SA on a computer. Such a proof checking system will not only check if an alleged proof (which is a sexp) is in fact a correct proof, but also will assist in constructing a formal proof. We believe that without such assistance by a computer, it will be impossible to actually construct a formal proof of a reasonably interesting theorem. The proof checking system will be implemented on Qute. As we have remarked in Section 2.1, Qute is a PROLOG-like language which computes functions and relations on sexps, and we can use Qute as a theorem prover for formal systems. Moreover, we can define the semantics of Quto formally within SA, so that it will be possible to prove properties of programs written in Qute. These topics, however, we leave for future publications.

Acknowledgements

The author is indebted to Dr. Susumu Hayashi for conversations on the material presented in this paper and to Mr. Takafumi Sakurai for implementing Qute.

References

[1] Beeson, M. J., Proving programs and programming proofs, *Abstracts of the 7th International Congress of Logic, Methodology and Philosophy of Science*, vol 1, (1983). 3–6,.

[2] Bourne, S. R., *The Unix System*, Addison–Wesley, 1982.

[3] de Bruijn, N. G., Lambda calculus notation with nameless dummies, A tool for automatic formula manipulation, with application to the Church–Rosser theorem, *Indag. Math.*, 34 (1972) 381–392.

[4] Feferman, S., Inductively presented system and formalization of meta-mathematics, *Logic Colloquium* '80, North-Holland, 1982.

[5] Gödel, K., Uber formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, *Monatshefte f. Math. u. Physik*, 38 (1931) 173–198.

[6] Gordon, M., Milner, R. and Wadsworth, C., *Edinburgh LCF*, Lect. Notes in Comp. Sci. 78, Springer 1978.

[7] Hayashi, S., Extracting Lisp programs from constructive proofs: A formal theory of constructive mathematics based on Lisp, *Publ. RIMS, Kyoto Univ.* 19 (1983) 169–191.

[8] McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, Part I, *Comm. ACM*, 3 (1960) 184–195.

[9] Quine, W. V., *Mathematical Logic* (*revised version*), Harvard University Press, 1979.

[10] Sato, M., Theory of symbolic expressions, I, *Theoretical Computer Science*, 22 (1983) 19–55.

[11] Sato, M. and Hagiya, M., Hyperlisp, in: J. W. de Bakker and J. C. van Vilet, Eds., *Algorithmic Languages*, North-Holland, (1981). 251–269.

[12] Sato, M. and Sakurai, T., Qute: A Prolog/Lisp type language for logic programming, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 507–513. 1983.

[13] Smullyan, R., *Theory of Formal System*, Annals of Mathematics Studies, 47, Princeton University Press, Princeton, 1961.