Assigning Meanings to Symbolic Objects – A Constructive Theory of Objects –

Masahiko Sato
Graduate School of Informatics
Kyoto University
masahiko@kuis.kyoto-u.ac.jp

Abstract

We present a constructive theory of objects in which a Turing complete functional programming language, named Z, can be formally specified. Our theory is developed in two stages. In the first stage, we introduce a set of symbolic expressions generated from two initial objects by two binary operators. This set is used as the meta-level universe in which the object language defined in the second stage is interpreted. In the second stage, we define the syntax of Z, by inductively defining a set of symbolic objects. Z is Lisp-like in the sense that both data and programs of Z are symbolic objects. We give both operational and denotational semantics to Z by interpreting Z data and programs as symbolic expressions.

1. Introduction

The purpose of this paper is (1) to present a simple theory of *symbolic expressions* by defining the class ⟨Sexpr⟩ of symbolic expressions, (2) to develop a theory of *symbolic objects* by defining the class ⟨Object⟩ of symbolic objects, and (3) to define a functional programming language Z by assigning meanings to symbolic objects. Both operational and denotational semantics will be given to Z by using ⟨Sexpr⟩ as the semantic domain to interpret the Z programs, i.e., symbolic objects.

All members of ⟨Sexpr⟩ and ⟨Object⟩ are *created* following the principle which we call the *fundamental principle of object creation*:

> Every object o is created from already created n objects $o_1, \ldots, o_n$ $(n \geq 0)$ by applying a *creation method* M.

We can visualize this *act* of creation by the following figure:

$$\frac{o_1 \quad \cdots \quad o_n}{o}\ \mathsf{M}$$

or, by the equation:

$$o = \mathsf{M}(o_1, \ldots, o_n)$$

As these visualizations suggest, we consider the method M as a *rule* and also as a *function*. Here, we require that each creation method must satisfy the *freeness conditions*, that is, (1) the method must be *injective*, and (2) objects created by the method must be *disjoint* from objects created by any other method. The first condition is a local condition which depends only on the method while the second condition is a global condition which depends on *all* creation methods available for creating objects.

If we create objects following the fundamental principle augmented by the freeness conditions, then we can *uniquely* recover M, n and $o_1, \ldots, o_n$ by analyzing the created object o. This fact implies that any object carries in itself a complete history of how it is created. So, we define two objects to be *equal* if and only if they are created exactly in the same way, and this equality makes our theory an *intensional* one (§**??**). We can also attach the scheme of proof-by-induction based on our inductive creation of objects and use it to prove that every object enjoys a certain property. Similarly, we can *inductively* define functions on objects, that is, we can attach a value to an object $o = \mathsf{M}(o_1, \ldots, o_n)$ using the values attached to o_i.

Our fundamental principle is a natural generalization of the principle that is used to define the set of natural numbers inductively. However, as far as we know, the present work is the first systematic application of the principle to create basic data structures (including functions and classes) that are necessary to develop a constructive theory of objects.

In §**??**, we follow the fundamental principle to define the class ⟨Sexpr⟩ of *symbolic expressions* by means of 4 creation methods Nil, T, Cons and Snoc. Here, the *class* ⟨Sexpr⟩ is defined as the set of those objects that can be created by the 4 methods, and simultaneously, we define the *concept* Sexpr (we use 'Sexpr' as an abbreviation of 'symbolic expression') by stipulating that an object o *falls under*

the concept Sexpr (or, *o is an* Sexpr) iff *o* is a member (or, an instance) of the class ⟨Sexpr⟩ (which we write $o : \langle\text{Sexpr}\rangle$). Following Frege [**?**], we view a concept as a function which, when applied to an object, returns the judgement asserting that the object falls under the concept. In this view, for example, we can write $\text{Sexpr}(o)$ for the application of the concept Sexpr to an object o, and read it: o is an Sexpr.

In §**??**, we define the class ⟨Object⟩ of *symbolic objects*, as well as its subclasses necessary to define ⟨Object⟩. In order to satisfy the freeness conditions, the construction is done independently from the construction in §**??**. As in §**??**, a class c is defined simultaneously with a concept C and they are related by the property: for any symbolic object o, o is a C iff $o : \mathsf{c}$ holds. We will say that c the *associated class* of C. The concepts introduced in this section are selected so that they form a *minimal* collection of concepts necessary to make ⟨Object⟩ a Turing complete programming language we define in §**??**.

In §**??**, we give operational and denotational semantics of Z in the domain ⟨Sexpr⟩. Our denotational semantics is unique in that with each Z expression e we assign four denotations $[\![e]\!]_{\text{p}}$ (denotation of e as a program), $[\![e]\!]_{\text{d}}$ (as a datum), $[\![e]\!]_{\text{f}}$ (as a function) and $[\![e]\!]_{\text{c}}$ (as a class), where, depending on e some of these denotations may be undefined.

§**??** gives concluding remarks comparing our work with related works. Philosophical motivation for this work is also discussed there.

2. Symbolic Expressions

In this section, we define the class ⟨Sexpr⟩ of *symbolic expressions* by creating its members following the fundamental principle. The concept *Sexpr* is defined by saying that any member of ⟨Sexpr⟩ is an Sexpr. Now, we define the class ⟨Sexpr⟩ by the signature:

$$\begin{aligned}
\mathsf{Nil} &: \quad\quad\quad\quad \to \langle\text{Sexpr}\rangle \\
\mathsf{Cons} &: \langle\text{Sexpr}\rangle\ \langle\text{Sexpr}\rangle \to \langle\text{Sexpr}\rangle \\
\mathsf{T} &: \quad\quad\quad\quad \to \langle\text{Sexpr}\rangle \\
\mathsf{Snoc} &: \langle\text{Sexpr}\rangle\ \langle\text{Sexpr}\rangle \to \langle\text{Sexpr}\rangle
\end{aligned}$$

where each of the creation methods Nil and T creates a new Sexpr from zero already created Sexprs, and each of the creation methods Cons and Snoc creates a new Sexpr from two already created Sexprs. Have we really defined the class ⟨Sexpr⟩ and the concept Sexpr? No! We have only *specified* the signature of ⟨Sexpr⟩, written $\Sigma_{\langle\text{Sexpr}\rangle}$. So, we have to explicitly show the existence of 4 methods having the signature $\Sigma_{\langle\text{Sexpr}\rangle}$ and satisfying the freeness conditions (§**??**). We do this by *implementing* these methods as concrete methods of placing the 5 kinds of tokens '`[`', '`]`', '`(`', '`)`' and '`|`', from left to right, where we assume that there are infinite supply of tokens for each kind. We can implement these methods either by the rules:

$$\frac{}{\texttt{[]}}\ \mathsf{Nil} \qquad \frac{s \quad t}{\texttt{[}s\,\texttt{|}\,t\texttt{]}}\ \mathsf{Cons} \qquad \frac{}{\texttt{()}}\ \mathsf{T} \qquad \frac{s \quad t}{\texttt{(}s\,\texttt{|}\,t\texttt{)}}\ \mathsf{Snoc}$$

or, equivalently, by the defining equations:

$$\begin{aligned}
\mathsf{Nil}() &:= \texttt{[]}, \quad \mathsf{Cons}(s,t) := \texttt{[}s\,\texttt{|}\,t\texttt{]}, \\
\mathsf{T}() &:= \texttt{()}, \quad \mathsf{Snoc}(s,t) := \texttt{(}s\,\texttt{|}\,t\texttt{)}.
\end{aligned}$$

In the creation methods Cons and Snoc above, s and t must be Sexprs already created. It is easy to *verify* the freeness of these methods. We have thus completed the definitions of the class ⟨Sexpr⟩ and the concept Sexpr. The class we defined is *categorical* in the sense that any two algebraic systems having the given signature and satisfying the freeness conditions are isomorphic. Thus $\Sigma_{\langle\text{Sexpr}\rangle}$ specifies the class abstractly but uniquely (up to isomorphism).

We write $\mathbb{S}$ for the class ⟨Sexpr⟩ when we view it as a concrete set of token sequences we have just implemented, and call it the *universe of symbolic expressions*. Thus $\mathbb{S}$ becomes a concrete algebra having signature $\Sigma_{\langle\text{Sexpr}\rangle}$ and satisfying the domain equation:

$$\mathbb{S} = \mathsf{Nil}() + \mathsf{T}() + \mathsf{Cons}(\mathbb{S},\mathbb{S}) + \mathsf{Snoc}(\mathbb{S},\mathbb{S})$$

where all the 4 creating methods are injective. We will also call these 4 methods *constructors* of Sexprs. In order to analyze Sexprs constructed thus, we introduce *recognizers* and *selectors* as follows.

We introduce the four recognizers:

$$\begin{aligned}
\mathsf{Nil?} &: \langle\text{Sexpr}\rangle \to \langle\text{Sexpr}\rangle \\
\mathsf{Cons?} &: \langle\text{Sexpr}\rangle \to \langle\text{Sexpr}\rangle \\
\mathsf{T?} &: \langle\text{Sexpr}\rangle \to \langle\text{Sexpr}\rangle \\
\mathsf{Snoc?} &: \langle\text{Sexpr}\rangle \to \langle\text{Sexpr}\rangle
\end{aligned}$$

where each recognizer returns `()` (true) if the given Sexpr is created by the method in question and `[]` (false) otherwise. Following Lisp's convention, we write `t` for `()` and `nil` for `[]`. For example, we have $\mathsf{Cons?}(\texttt{[t|nil]}) = \texttt{t}$ and $\mathsf{Nil?}(\texttt{t}) = \texttt{nil}$.

As for selectors, we have:

$$\begin{aligned}
\mathsf{Car} &: \langle\text{Sexpr}\rangle \dashrightarrow \langle\text{Sexpr}\rangle \\
\mathsf{Cdr} &: \langle\text{Sexpr}\rangle \dashrightarrow \langle\text{Sexpr}\rangle \\
\mathsf{Head} &: \langle\text{Sexpr}\rangle \dashrightarrow \langle\text{Sexpr}\rangle \\
\mathsf{Tail} &: \langle\text{Sexpr}\rangle \dashrightarrow \langle\text{Sexpr}\rangle
\end{aligned}$$

We used the dash arrow symbol '$\dashrightarrow$' to indicate that these functions are partial functions from ⟨Sexpr⟩ to ⟨Sexpr⟩. The selectors Car and Cdr are used to select Sexprs that are used to create an Sexpr by the creation method Cons. That is,

$\mathsf{Car}(s)$ ($\mathsf{Cdr}(s)$) selects the first (second, respectively) Sexpr that is used to construct s and they are defined only when $\mathsf{Cons?}(s) = \mathtt{t}$. Similarly, the selectors Head and Tail are used to select Sexprs from an Sexpr created by the Snoc method. There are no selectors for `nil` and `t` since they are created from zero previously created Sexprs.

We have thus introduced 12 functions (4 constructors, 4 recognizers and 4 selectors) on $\langle$Sexpr$\rangle$. By adding the mechanism of conditional computation and the mechanism of defining a new function by recursion we can define the class of computable functions on $\langle$Sexpr$\rangle$ in an analogous way as we define computable functions on natural numbers.

Let us introduce a form of conditional computation:

$$e[a; b; c; d]$$

which is computed as follows. First, we compute e. If e does not have a value, then the value of the form is undefined. If e has a value, then e's value is either (1) `nil`, (2) a Consed object, (3) `t`, or (4) a Snoced object. We then compute a in case (1), b in case (2), c in case (3) and d in case (4). We call this form a *case-form*. We also introduce an *if-form* $e\langle a; b\rangle$ (read: if e then a else b) by putting:

$$e\langle a; b\rangle := e[b; a; a; a]$$

The value of this if-form is: the value of a if e has a non-`nil` value, the value of b if e has value `nil`, and undefined otherwise.

As an example of a recursive definition of a function, we define the function Eq? which decides for given two objects whether they are equal:

$$\begin{array}{l}
\mathsf{Eq?}(x, y) := \\
x[y\langle \mathtt{nil}; \mathtt{t}\rangle; \\
\quad y[\mathtt{nil}; \mathtt{nil}; \\
\quad\quad \mathsf{Eq?}(\mathsf{Car}(x), \mathsf{Car}(y))\langle \mathsf{Eq?}(\mathsf{Cdr}(x), \mathsf{Cdr}(y)); \mathtt{nil}\rangle; \\
\quad\quad \mathtt{nil}]; \\
\quad y\langle \mathtt{t}; \mathtt{nil}\rangle; \\
\quad y[\mathtt{nil}; \mathtt{nil}; \mathtt{nil}; \\
\quad\quad \mathsf{Eq?}(\mathsf{Head}(x), \mathsf{Head}(y))\langle \mathsf{Eq?}(\mathsf{Tail}(x), \mathsf{Tail}(y)); \mathtt{nil}\rangle]]
\end{array}$$

We have the following theorem which shows that Eq? computes the equality relation correctly.

Theorem 1. *For any Sexprs s and t, we have $\mathsf{Eq?}(s, t) = \mathtt{t}$ if and only if $s = t$.*

Proof. First we can show by the double induction on creations of s and t that Eq? is a total function and always return either `nil` or `t`. Then, again by the double induction on creations of s and t, we can show that (1) if $\mathsf{Eq?}(s, t) = \mathtt{t}$, then $s = t$ and (2) if $\mathsf{Eq?}(s, t) = \mathtt{nil}$, then $s \neq t$. □

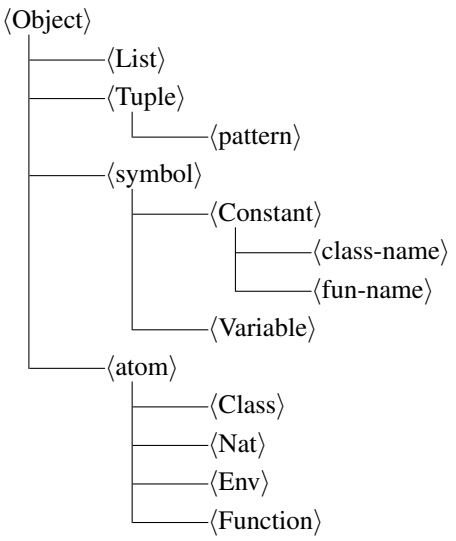

Figure 1. ⟨Object⟩ and its subclasses

We conclude this section by introducing a number of notational conventions that we will use in the coming sections.

- $\mathtt{[}s\mathtt{]} := \mathtt{[}s \mid \mathtt{[]}\mathtt{]}$
- $\mathtt{[}s_1 \cdots s_n\mathtt{]} := \mathtt{[}s_1 \mid \mathtt{[}s_2 \cdots s_n\mathtt{]}\mathtt{]} (n \geq 2)$
- $\mathtt{(}s\mathtt{)} := \mathtt{(}s \mid \mathtt{()}\mathtt{)}$
- $\mathtt{(}s_1 \cdots s_n\mathtt{)} := \mathtt{(}s_1 \mid \mathtt{(}s_2 \cdots s_n\mathtt{)}\mathtt{)} (n \geq 2)$

We also abbreviate $\mathtt{[}s \mid \mathtt{[}t \mid u\mathtt{]}\mathtt{]}$ $\mathtt{(}\mathtt{(}s \mid \mathtt{(}t \mid u\mathtt{)}\mathtt{)}\mathtt{)}$ to $\mathtt{[}s\ t \mid u\mathtt{]}$ $\mathtt{(}\mathtt{(}s\ t \mid u\mathtt{)}\mathtt{)}$, respectively), etc.

3. Symbolic Objects

In this section, we define the class ⟨Object⟩ of *symbolic objects* together with its 13 subclasses. Figure 1 shows the subclass relation among these 14 classes. We call a class *basic class* if it is the collection of all the objects which can be created by one or more specified creation methods. Thus a basic class is defined by specifying its creation methods. A non-basic class is called a *derived class* and it is defined either as a union of two or more classes or as a subclass of another class. When we name a class, we follow the convention of capitalizing the first letter of the name of a basic class and use lower-case letters for a derived class[1]. So, for example, ⟨List⟩ is a basic class and ⟨symbol⟩ is a derived class.

As in §**??**, classes and concepts are defined in three steps: specification, implementation and verification. However, to save space, we skip the specification step here[2] and directly

[1] An exception is the derived class ⟨Object⟩. We capitalize the initial letter since this class forms the universe of symbolic objects.

[2] See Appendix **??** for the specification.

implement the specified classes as a set of sequences of the five tokens we used in §**??**. The verification that our implementation satisfies the freeness conditions is easy and we remark on this only in §**??**. This implementation determines the concrete syntax of our programming language Z.

3.1. Object

We define the class ⟨Object⟩ as the union of 4 disjoint classes:

$$\langle\text{Object}\rangle := \langle\text{Tuple}\rangle + \langle\text{List}\rangle + \langle\text{symbol}\rangle + \langle\text{atom}\rangle$$

where ⟨symbol⟩ is defined as the union of 2 disjoint basic classes (§**??**) and ⟨atom⟩ is defined as the union of 4 disjoint basic classes (§**??**). Thus ⟨Object⟩ becomes the union of 8 disjoint basic classes. We define the *mother class* of an Object as the unique basic class to which the Object belongs.

We write $\mathbb{U}$ for the class ⟨Object⟩ when we view it as a concrete set of token sequences, and call it the *universe of symbolic objects*. Actually, we implement ⟨Object⟩ in such a way that each Object as a token sequence belongs to the set $\mathbb{S}$. Thus, we will have $\mathbb{U} \subseteq \mathbb{S}$. Note that this is an accidental fact that depends on our particular implementation of ⟨Sexpr⟩ and ⟨Object⟩. Hence, this fact does not mean that ⟨Object⟩ is a subclass of ⟨Sexpr⟩. Rather, these two classes are conceptually independent, and we could implement these two classes so that they become mutually disjoint.

As we are following the fundamental principle, any Object o is created by its creation method from already created Objects: $o = M\,(o_1, \ldots, o_n)$. Here, we call M the *creation method* of o, and o_i $(1 \le i \le n)$ the i-th *component* of o[3]. In order to be able to completely and uniformly analyze o, we introduce the 5 primitive functions below:

$$\begin{aligned}
\mathsf{method} &: \langle\text{Object}\rangle \to \langle\text{fun-name}\rangle \\
\mathsf{1st} &: \langle\text{Object}\rangle \dashrightarrow \langle\text{Object}\rangle \\
\mathsf{2nd} &: \langle\text{Object}\rangle \dashrightarrow \langle\text{Object}\rangle \\
\mathsf{3rd} &: \langle\text{Object}\rangle \dashrightarrow \langle\text{Object}\rangle \\
\mathsf{=?} &: \langle\text{Object}\rangle\ \langle\text{Object}\rangle \to \langle\text{Object}\rangle
\end{aligned}$$

The function method computes the creation method M of o. As M itself can never be an Object (a method is something the operates on Objects at meta-level) the function method cannot directly return M, so it returns instead the *name* of the creation method M (see §**??**). The functions 1st, 2nd and 3rd respectively compute the first, second and third component of o. Finally, =? is a binary function which decides whether given two Objects are equal or not. Thus =? returns `()` (true) if they are equal, and `[]` (false) otherwise.

[3] In our construction of ⟨Object⟩, n is at most 3.

Just as we could define all the computable functions on ⟨Sexpr⟩ by the 12 primitive functions defined in §**??**, we can define all the computable functions on ⟨Object⟩ using these 5 primitive functions and the 14 creation methods we introduce in §**??**–§**??**.

3.2. List

We define the class ⟨List⟩ as a basic class whose members are created by the methods:

$$\frac{}{\texttt{[]} : \langle\text{List}\rangle}\ \mathsf{nil} \qquad \frac{a : \langle\text{Object}\rangle \quad b : \langle\text{List}\rangle}{\texttt{[}a \mid b\texttt{]} : \langle\text{List}\rangle}\ \mathsf{cons}$$

We have the following recognizers and selectors.

$$\begin{aligned}
\mathsf{nil?} &: \langle\text{Object}\rangle \to \langle\text{Object}\rangle \\
\mathsf{cons?} &: \langle\text{Object}\rangle \to \langle\text{Object}\rangle \\
\mathsf{car} &: \langle\text{List}\rangle \dashrightarrow \langle\text{Object}\rangle \\
\mathsf{cdr} &: \langle\text{List}\rangle \dashrightarrow \langle\text{List}\rangle
\end{aligned}$$

The recognizer cons?, for instance, is defined by:

$$\mathsf{cons?}(x) := \mathsf{=?}(\mathsf{method}(x), \texttt{cons})$$

where `cons` is the function name (§**??**) for the function cons. This recognizer decides if x is created by the creation method cons. We can similarly define the recognizer M? for every creation method M. Using these recognizers, we can define the recognizer C? for every class C. For instance, the recognizer List? : ⟨Object⟩ → ⟨Object⟩ is defined by:

$$\mathsf{List?}(x) := \mathsf{or}(\mathsf{nil?}(x), \mathsf{cons?}(x))$$

where or decides if one of its arguments is `()`. The selector car is defined by 1st but differs from 1st in that car is defined only on ⟨List⟩. For example, car of a Tuple $(a \mid b)$ (§**??**) is undefined (since a Tuple is not a List) while 1st returns a. cdr is defined similarly.

3.3. Tuple

The class ⟨Tuple⟩ is defined as a basic class whose structure is isomorphic to the structure of ⟨List⟩. Although ⟨Tuple⟩ and ⟨List⟩ share the same mathematical structure, it is useful to have both classes and distinguish them conceptually. In general, we will use Tuples to encode Cartesian products of several classes, and use Lists to encode sequences of the the same or similar classes.

The creation methods for ⟨Tuple⟩ are:

$$\frac{}{\texttt{()} : \langle\text{Tuple}\rangle}\ \mathsf{t} \qquad \frac{a : \langle\text{Object}\rangle \quad b : \langle\text{Tuple}\rangle}{(a \mid b) : \langle\text{Tuple}\rangle}\ \mathsf{snoc}$$

We have the following functions for tuples.

$$\begin{aligned} \mathsf{t?} &: \langle\text{Object}\rangle \to \langle\text{Object}\rangle \\ \mathsf{snoc?} &: \langle\text{Object}\rangle \to \langle\text{Object}\rangle \\ \mathsf{head} &: \langle\text{Tuple}\rangle \dashrightarrow \langle\text{Object}\rangle \\ \mathsf{tail} &: \langle\text{Tuple}\rangle \dashrightarrow \langle\text{Tuple}\rangle \end{aligned}$$

3.4. Symbol

The derived class ⟨symbol⟩ is defined as the union of two disjoint basic classes ⟨Constant⟩ and ⟨Variable⟩.

3.5. Constant

The basic class ⟨Constant⟩ is defined by the following creation method:

$$\frac{o : \langle\text{Object}\rangle}{\texttt{[}o\texttt{|(())]} : \langle\text{Constant}\rangle}\ \mathsf{constant}$$

We introduce a notational convention for expressing some Constants in `typewriter` font. For example, the convention allows us to write `ab` for the Constant `[[(()()[][][][]())(()()[][][]()[])]|()]`. Here, we used 7-bit ASCII code for encoding ASCII characters.

3.6. Variable

The basic class ⟨Variable⟩ is defined by:

$$\frac{o : \langle\text{Object}\rangle}{\texttt{[}o\texttt{|((()))]} : \langle\text{Variable}\rangle}\ \mathsf{variable}$$

As a notational convention, we will write a Variable in *slant* typewriter font. For example, x denotes the Variable `[(()()()()[][][])|(())]`.

3.7. Atom

The class ⟨atom⟩ is defined as the union of 4 disjoint basic classes:

$$\langle\text{atom}\rangle := \langle\text{Class}\rangle + \langle\text{Nat}\rangle + \langle\text{Env}\rangle + \langle\text{Function}\rangle$$

Each subclass C of ⟨atom⟩ will be defined in such a way that any Object o in C will be of the form `(`a`|`c`)` where c is a Constant which is the class-name (§**??**) of C, and a is an Object which together with c enables us to recover from o the components and the creation method of o. It then follows that all subclasses of ⟨atom⟩ are mutually disjoint as well as that ⟨atom⟩ is disjoint from ⟨List⟩, ⟨Tuple⟩ and ⟨symbol⟩.

3.8. Class

A Constant is said to be a **class-name** if it is `<Object>`, `<List>`, . . . , or `<Function>`, namely, if it corresponds to a name of the 14 classes listed in Figure 1.

The class ⟨Class⟩ is a basic class and it has the creation method class:

$$\frac{c : \langle\text{class-name}\rangle}{\texttt{(}c\texttt{|<Class>)} : \langle\text{Class}\rangle}\ \mathsf{class}$$

This method creates a Class which, in Z, internalizes the (meta-level) class whose class-name is c. So, we can define the following function:

$$\mathsf{in?} : \langle\text{Object}\rangle\ \langle\text{Class}\rangle \to \langle\text{Object}\rangle$$

with the property: $\mathsf{in?}(o, \texttt{(}c\texttt{|<Class>)}) = \texttt{t}$ iff $o : \mathsf{C}$, where C is the meta-level class whose class-name is c.

3.9. Natural Number

The class ⟨Nat⟩ is defined by the two creation methods:

$$\frac{}{\texttt{(zero|<Nat>)} : \langle\text{Nat}\rangle}\ \mathsf{zero}$$

$$\frac{\texttt{(}n\texttt{|<Nat>)} : \langle\text{Nat}\rangle}{\texttt{((succ }n\texttt{)|<Nat>)} : \langle\text{Nat}\rangle}\ \mathsf{succ}$$

We use decimal notation to denote Nats. For instance, $0 =$ `(zero|<Nat>)` and $1 =$ `((succ zero)|<Nat>)`.

We have the following selector for natural numbers:

$$\mathsf{pred} : \langle\text{Nat}\rangle \dashrightarrow \langle\text{Nat}\rangle$$

which returns the predecessor of n for $n \neq 0$.

3.10. Environment

We define the basic class ⟨Env⟩ by the following two creation methods:

$$\frac{}{\texttt{([]|<Env>)} : \langle\text{Env}\rangle}\ \mathsf{empty}$$

$$\frac{s : \langle\text{symbol}\rangle \quad v : \langle\text{Object}\rangle \quad \texttt{(}E\texttt{|<Env>)} : \langle\text{Env}\rangle}{\texttt{([(}s\ v\texttt{)|}E\texttt{]|<Env>)} : \langle\text{Env}\rangle}\ \mathsf{put}$$

For the sake of notational convenience, we abbreviate `([`$b_1\ \cdots\ b_n$`]|<Env>)` to $\{b_1\ \cdots\ b_n\}$, where each b_i is a pair of a symbol and an Object. We also write $E[v/s]$ for $\mathsf{put}(s, v, E)$.

An Env is *global* (*local*) if each s_i is a Constant (Variable, respectively). We use the meta-variables Γ and Δ for global Envs and Λ and Π for local Envs. A pair of global and local Envs is used to provide a context in which Constants and Variables are evaluated (§**??**).

We define the following functions:

$$\begin{aligned} \mathsf{get} &: \langle \text{symbol} \rangle\ \langle \text{Env} \rangle \dashrightarrow \langle \text{Object} \rangle \\ \mathsf{conc} &: \quad \langle \text{Env} \rangle\ \langle \text{Env} \rangle \ \to \langle \text{Env} \rangle \end{aligned}$$

The value of $\mathsf{get}(s, E)$ is defined only if E contains a Tuple of the form `(`s v`)`, and in this case, get returns the v of the leftmost such Tuple. We will write $E(s)$ for $\mathsf{get}(s, E)$. We define the *size* of an Env E, written $|E|$, to be the number of symbols s for which $E(s)$ is defined. The function conc concatenates two Envs in an obvious way. For instance, we have $\mathsf{conc}(\{a\ b\}, \{c\ d\}) = \{a\ b\ c\ d\}$. We write E, E' for $\mathsf{conc}(E, E')$.

3.11. Pattern

A Tuple d is said to be a *declaration* if it is of the form x, `(`c x`)` or of the form `(&rest` c x`)` where c is a class-name (§**??**) and x is a Variable. If d is of the third form, it is called a *special declaration*. A **pattern** is defined to be a Tuple of the form `(`d_1 $\cdots$ d_n`)` where each d_i is a declaration.

A pattern is used to match a Tuple against it. So we define the binary partial function

$$\mathsf{match} : \langle \text{Tuple} \rangle\ \langle \text{pattern} \rangle \dashrightarrow \langle \text{Env} \rangle$$

which matches a Tuple e against a pattern p, and returns an Env if the matching succeeds. For the pattern $p =$ `((<List> x) (<Nat> y))`, for instance, we have $\mathsf{match}($`([] 1)`$, p) =$ `{(x []) (y 1)}`. On the other hand, $\mathsf{match}($`(() 1)`$, p)$ is undefined since `()` is not a ⟨list⟩. But we have $\mathsf{match}($`((),(x)`$) =$ `{(x ())}` since the declaration x is treated as an abbreviation of `(<Object> x)`. Just as in Lisp, a special declaration is used to bind a Tuple of Objects to a single Variable, but we skip the details.

3.12. Function

The class ⟨Function⟩ is a basic class and it has the creation methods function, defun and defmacro.

$$\frac{c : \langle \text{fun-name} \rangle}{\texttt{(}c\texttt{ |<Function>)} : \langle \text{Function} \rangle}\ \mathsf{function}$$

In this method, c must be a **fun-name** (function name), that is, it must be one of the following Constants:

```
set! defun defmacro progn let if quote
method 1st 2nd 3rd =? cons snoc
constant variable class zero succ
empty put function
```

The first 7 fun-names in the above list create macro Functions and the remaining 14 fun-names create call-by-value Functions.

The creation method defun is defined by the rule:

$$\frac{g : \langle \text{Constant} \rangle \quad p : \langle \text{pattern} \rangle \quad b : \langle \text{Tuple} \rangle}{\texttt{((f }g\ p\texttt{ |}b\texttt{) |<Function>)} : \langle \text{Function} \rangle}\ \mathsf{defun}$$

This rule, as well as the next rule for defmacro, must satisfy a special side condition on g. We explain the condition in Case 1.2 and Case 1.3 of §**??**. The Tuple b is the body of the defined Function. After the method is invoked, the value of the Constant g becomes the Function created by defun (§**??**). So, the method defun not only creates a new Function but also assigns the name g to the created function.

The creation method defmacro is defined by the rule:

$$\frac{g : \langle \text{Constant} \rangle \quad p : \langle \text{pattern} \rangle \quad b : \langle \text{Tuple} \rangle}{\texttt{((m }g\ p\texttt{ |}b\texttt{) |<Function>)} : \langle \text{Function} \rangle}\ \mathsf{defmacro}$$

In §**??**, we explain how we can dynamically extend the open-ended universe ⟨Object⟩ using defun and defmacro.

4. Meanings of Symbolic Objects

In §**??** we defined the universe of symbolic objects $\mathbb{U}$ as a subset of the universe of symbolic expressions $\mathbb{S}$. In this section we assign meanings to Objects by treating Objects as Sexprs and by *viewing* them as *programs*, *data*, *functions* and *classes*.

First we give a meaning of the Z language by viewing each Object as a program. To this end, we define a binary *evaluation* relation $e \Downarrow v$ (read: e evaluates to v, or e has value v, or e denotes v) between two Objects e (which we call a *program*) and v (which we call a *value*). To be more precise, the evaluation of a program is done in a context determined by two Envs of which one is global and the other is local. So the evaluation relation depends on these two Envs. Moreover, the context itself sometimes changes after the evaluation since our language supports dynamic extension of the universe by Function definitions.Thus, in general, an evaluation relation is not a binary relation but a relation on 6 Objects, which we write:

$$\Gamma; \Lambda \vdash e \Downarrow v \dashv \Delta; \Pi$$

where $\Gamma; \Lambda$ is the context before the evaluation and $\Delta; \Pi$ is the context after the evaluation. However, we will continue to express the relation as a binary relation leaving Envs implicit. Or, we will write $\Gamma \vdash e \Downarrow v \dashv \Delta$ ($\Lambda \vdash e \Downarrow v \dashv \Pi$) when we can leave local (global, resp.) Envs implicit. We define the relation $e \Downarrow v$ inductively on how e is created.

A List is evaluated by the two rules below:

$$\frac{}{\texttt{[]} \Downarrow \texttt{[]}} \qquad \frac{a \Downarrow u \quad b \Downarrow v}{\texttt{[}a\texttt{ |}b\texttt{]} \Downarrow \texttt{[}u\texttt{ |}v\texttt{]}}$$

In order to guarantee the deterministic evaluation (Theorem **??**), we understand that the premises of each evaluation rule

are evaluated from left to right. Thus the second rule above, written in full, is:

$$\frac{\Gamma;\Lambda \vdash a \Downarrow u \dashv \Gamma';\Lambda' \quad \Gamma';\Lambda' \vdash b \Downarrow v \dashv \Gamma'';\Lambda''}{\Gamma;\Lambda \vdash \texttt{[}a \texttt{|} b\texttt{]} \Downarrow \texttt{[}u \texttt{|} v\texttt{]} \dashv \Gamma'';\Lambda''}$$

A symbol is evaluated by retrieving its value from the current global or local Env. The value is undefined if the value is not found in the Env.

$$\frac{c : \langle\text{Constant}\rangle \quad \Gamma(c) = v}{\Gamma \vdash c \Downarrow v \dashv \Gamma} \qquad \frac{x : \langle\text{Variable}\rangle \quad \Lambda(x) = v}{\Lambda \vdash x \Downarrow v \dashv \Lambda}$$

An atom evaluates to itself.

$$\frac{e : \langle\text{atom}\rangle}{e \Downarrow e}$$

From Figure 1, we see that the only remaining case to be covered by the evaluation rules is the case where the program to be evaluated is a Tuple. To cover this case, it is necessary to introduce an auxiliary ternary relation $e \mapsto_f v$ (read: the result of *applying* f to its *argument* e is v, or f maps e to v) among a Function f, a Tuple e and an Object v. This relation also depends on contexts which we leave implicit as much as possible. Using this application relation, we can define the following two evaluation rules for a Tuple:

$$\frac{}{\texttt{()} \Downarrow \texttt{()}} \qquad \frac{f \Downarrow g \quad g : \langle\text{Function}\rangle \quad e \mapsto_g v}{\texttt{(}f \texttt{|} e\texttt{)} \Downarrow v}$$

The first rule is for evaluating `()`. The second is for evaluating a Tuple which is not `()`. This rule says that $\texttt{(}f \texttt{|} e\texttt{)}$ has value v if and only if f evaluates to a Function g and the result of applying g to e is v.

The application relation $e \mapsto_f v$ is defined by case analysis (case 1 to case 4 below) on the creation of Function f.

Case 1: In this case we give evaluation rules for the primitive call-by-name Functions we introduced in §**??**. The first 3 functions (1.1–1.3) are used to change the surrounding contexts and the values returned by these functions are less significant.

1.1 The rules for $f =$ `(set!|<Function>)` are:

$$\frac{c : \langle\text{Constant}\rangle \quad \Gamma \vdash a \Downarrow v \dashv \Delta}{\Gamma \vdash \texttt{(}c\ a\texttt{)} \mapsto_f v \dashv \Delta[v/c]} \qquad \frac{x : \langle\text{Variable}\rangle \quad \Lambda \vdash a \Downarrow v \dashv \Pi}{\Lambda \vdash \texttt{(}x\ a\texttt{)} \mapsto_f v \dashv \Pi[v/x]}$$

In the left rule above, c must be a Constant which does not have a value in Δ. In the right rule above, on the other hand, x must be a Variable which does have a value in Π. We have $|\Delta| = |\Gamma| + 1$ and $|\Pi| = |\Lambda|$ when these rules are applied.

1.2 The rule for $f =$ `(defun|<Function>)` is:

$$\frac{g : \langle\text{Constant}\rangle \quad p : \langle\text{pattern}\rangle \quad b : \langle\text{Tuple}\rangle}{\Gamma \vdash \texttt{(}g\ p \texttt{|} b\texttt{)} \mapsto_f \texttt{()} \dashv \Gamma\big[\texttt{((f}\ g\ p \texttt{|} b\texttt{)} \texttt{|<Function>)}/g\big]}$$

where g must be a Constant whose value is undefined in Γ. Note that when g satisfies this side-condition, we can invoke the (meta-level) method defun and can extend the universe $\mathbb{U}$ dynamically by adding a new Function to the universe.

1.3 The rule for $f =$ `(defmacro|<Function>)` is:

$$\frac{g : \langle\text{Constant}\rangle \quad p : \langle\text{pattern}\rangle \quad b : \langle\text{Tuple}\rangle}{\Gamma \vdash \texttt{(}g\ p \texttt{|} b\texttt{)} \mapsto_f \texttt{()} \dashv \Gamma\big[\texttt{((m}\ g\ p \texttt{|} b\texttt{)} \texttt{|<Function>)}/g\big]}$$

where g must be a Constant whose value is undefined in Γ.

1.4 The rules for $f =$ `(progn|<Function>)` are:

$$\frac{a \Downarrow v}{\texttt{(}a\texttt{)} \mapsto_f v} \qquad \frac{a \Downarrow u \quad b \mapsto_f v}{\texttt{(}a \texttt{|} b\texttt{)} \mapsto_f v}$$

In the second rule, b must be a non-`()` Tuple. The Function `progn` evaluates members of its argument one by one and return the value of the last member.

1.5 To define the rule for $f =$ `(let|<Function>)`, we introduce an auxiliary binary relation $\Pi \Downarrow_{\text{Env}} \Pi'$ on local Envs as follows.

$$\frac{\Pi : \langle\text{Env}\rangle \quad a_1 \Downarrow v_1 \quad \cdots \quad a_n \Downarrow v_n}{\Pi \Downarrow_{\text{Env}} \Pi'}$$

where $\Pi = \{\,\texttt{(}x_1\ a_1\texttt{)} \cdots \texttt{(}x_n\ a_n\texttt{)}\,\}$. and $\Pi' = \{\,\texttt{(}x_1\ v_1\texttt{)} \cdots \texttt{(}x_n\ v_n\texttt{)}\,\}$ $(n \geq 0)$.

Using this relation, we give the rule for $f =$ `(let|<Function>)` as follows.

$$\frac{\Lambda \vdash \Pi \Downarrow_{\text{Env}} \Pi' \dashv \Lambda' \quad \Pi', \Lambda' \vdash e \mapsto_g v \dashv \Pi'', \Lambda''}{\Lambda \vdash \texttt{(}\Pi \texttt{|} e\texttt{)} \mapsto_f v \dashv \Lambda''}$$

where $g =$ `(progn|<Function>)` and Λ' (Π') and Λ'' (Π'', resp.) must be of the same length. Here, Π is used to create a temporarily extended local Env Π', Λ', and the body e of the `let`-form is evaluated in the extended local Env. The extended part of the Env is thrown away after the evaluation.

1.6 The rules for $f =$ `(if|<Function>)` are:

$$\frac{c \Downarrow \texttt{()} \quad a \Downarrow v}{\texttt{(}c\ a\ b\texttt{)} \mapsto_f v} \qquad \frac{c \Downarrow \texttt{[]} \quad b \Downarrow v}{\texttt{(}c\ a\ b\texttt{)} \mapsto_f v}$$

1.7 The rule for $f =$ `(quote|<Function>)` is:

$$\frac{}{\texttt{(}e\texttt{)} \mapsto_f e}$$

Case 2: We now give rules for call-by-value primitive Functions. In call-by-value Function application, a Function's argument is evaluated before application. To define the rule for this case, we introduce two auxiliary binary relations. The first one is $e \Downarrow_{\text{Tup}} v$ on Tuples defined as follows.

$$\frac{e_1 \Downarrow v_1 \quad \cdots \quad e_n \Downarrow v_n}{\texttt{(}e_1 \cdots e_n\texttt{)} \Downarrow_{\text{Tup}} \texttt{(}v_1 \cdots v_n\texttt{)}} \ (n \geq 0)$$

The second relation $e \rhd_f v$ is a ternary relation among a Function f, Tuple e and Object v. The relation is defined as follows.

$$\frac{\mathsf{f}(v_1, \ldots, v_n) = v}{(v_1 \ \ldots \ v_n) \rhd_f v} \ (n \geq 0)$$

where f is the meta-level function that corresponds to f whose computation rule we gave in §**??**. For example, if f is `(car|<Function>)`, then f is car defined in §**??**.

Now, the general rule for such a Function f is:

$$\frac{e \Downarrow_{\mathrm{Tup}} u \quad u \rhd_f v}{e \mapsto_f v}$$

Case 3 $f =$ `((f g p|b)|<Function>)`**:** This is the case where f is defined by the defun method.

$$\frac{\Lambda \vdash e \Downarrow_{\mathrm{Tup}} u \dashv \Lambda' \quad \mathsf{match}(u, p) = \Pi \quad \Pi \vdash b \mapsto_h v \dashv \Pi'}{\Lambda \vdash e \mapsto_f v \dashv \Lambda'}$$

where $h =$ `(progn|<Function>)`. Here, the argument e is evaluated as a Tuple in the surrounding local Env Λ, and the result is used to create a local Env Π. Then, the body b of f is evaluated in Π.

Case 4 $f =$ `((m g p|b)|<Function>)`**:** This is the case where f is defined by the defmacro method. In this case f is intended to work as a macro (in the sense of Lisp).

$$\frac{\mathsf{match}(e, p) = \Pi \quad \Pi \vdash b \mapsto_h b' \dashv \Pi' \quad \Lambda \vdash b' \Downarrow v \dashv \Lambda'}{\Lambda \vdash e \mapsto_f v \dashv \Lambda'}$$

where $h =$ `(prog|<Function>)`. Here, the body b of f is evaluated in the temporal local Env Π, and the result b' is evaluated in the surrounding local Env Λ.

We have thus completed the definition of the evaluation relation. We now give a few examples of evaluation in Z. If the global Env is $\Gamma_0 = \{\}$ in which no Constant has a value, what would be the first program we wish to evaluate? Our recommendation is:

```
((set!|<Function>) set! (set!|<Function>))
```

By evaluating this program, the Constant `set!` is set to have the function $\mathsf{s} =$ `(set!|<Function>)` as its value, and we get a new Env Γ_1 in which we have `set!` $\Downarrow \mathsf{s}$. So in Γ_1 (and afterwards) we can use `set!` as a shorthand for s, and both `(set!` x a`)` and `(`s x a`)` have the same value (if any) for any a. We can continue to set values for Constants as we like in this way. In particular, we set `quote` to `(quote|<Function>)` and, as in Lisp, write $'e$ for `(quote` e`)`. We also set values to Constants which correspond to primitive functions and classes as well as `[]` to `nil` and `()` to `t`.

Having done this, we can extend our universe by `defmacro` and `defun` as shown in Figure 2. There, we first define the macro Function `while`. Then, after defining

```
(defmacro tuple (&rest a)
  (if (t? a) a
    (snoc snoc
     (snoc (head a)
      (snoc (snoc tuple (tail a)) t)))))
(defmacro while (c &rest b)
  (tuple if c
         (tuple progn (snoc progn b)
                (snoc while (snoc c b)))
         nil))
(defun + ((<nat> m) (<nat> n))
  (if (=? n 0) m (succ (+ m (pred n)))))
(defun * ((<nat> m) (<nat> n))
  (if (=? n 0) 0 (+ (* m (pred n)) m)))
(defun factr ((<nat> n))
  (if (=? n 0) 1 (* n (factr (pred n)))))
(defun facti ((<nat> n))
  (let {(result 1)}
    (while (not (=? n 0))
      (set! result (* n result))
      (set! n (pred n)))
    result))
```

Figure 2. Example of Function Definitions

addition (+) and multiplication ($\star$) on $\langle$Nat$\rangle$, the factorial Function is defined by recursion (`factr`) and by iteration (`facti`). All of these Functions were tested on a prototype Z interpreter which we implemented in Emacs Lisp. The evaluation relation enjoys the following properties.

Theorem 2. *If* $\Gamma; \Lambda \vdash e \Downarrow v \dashv \Delta; \Pi$ *and* $\Gamma; \Lambda \vdash e \Downarrow v' \dashv \Delta'; \Pi'$*, then* $v = v'$*,* $\Delta = \Delta'$*,* $\Pi = \Pi'$*,* $|\Gamma| \leq |\Delta|$ *and* $|\Lambda| = |\Pi|$.

Corollary 1. *If* $\Gamma; \{\} \vdash e \Downarrow v \dashv \Delta; \Pi$*, then* $\Pi = \{\}$.

Theorem **??** shows that evaluation is deterministic. Corollary **??** characterizes the top-level of a Z interpreter. Namely, the interpreter sets the local Env to be empty before the evaluation of a program, and only the global Env Γ might change to Δ after the evaluation either by the `set!` command on a Constant, by the `defun` command, or by the `defmacro` command. The interpreter then evaluates the next program input by the user in the new global Env Δ.

We can see this cycle of evaluation as a process of dynamically changing the universe $\mathbb{U}$ of symbolic objects. To analyze this process, we take advantage of the fact that we have implemented $\mathbb{U}$ as a subset of $\mathbb{S}$. So, we will work in $\mathbb{S}$ and see how $\mathbb{U}$ can be extended within $\mathbb{S}$. We say that a global Env Δ is *realizable* if either (1) $\Delta = \{\}$ or (2) $\Delta = \Gamma[o/c]$, where Γ is realizable, c is a Constant, and Δ is obtained by applying a function f in Case 1.1–1.3. So, a realizable Env can actually be obtained by evaluating Z programs appropriately. It is easy to see that we can decide, for any Sexpr, if it is a realizable Env.

Let Γ be a realizable Env. We say that a Function f is a Γ-*defined Function* if $f = \Gamma(g)$ for some Constant g. We define $\mathbb{U}_\Gamma$ to be the smallest subset X of $\mathbb{S}$ such that (1) X contains all the Γ-defined Functions and (2) X is closed under all the creation methods other than defun and defmacro. It is easy to see that $\mathbb{U}_\Gamma$ is a decidable subset of $\mathbb{S}$. We define the partial function:

$$\mathsf{eval}_\Gamma : \mathbb{U}_\Gamma \dashrightarrow \mathbb{S}$$

by putting $\mathsf{eval}_\Gamma(e) = v$ iff $\Gamma; \{\} \vdash e \Downarrow v \dashv \Gamma'; \{\}$ for some Γ'. We also write $\Gamma[\![e]\!]_\mathrm{p}$ for $\mathsf{eval}_\Gamma(e)$. $\Gamma[\![e]\!]_\mathrm{p}$ gives the denotation (in $\mathbb{S}$) of e viewed as a **program**.

Let us now turn to other views of Z Objects. We can view an Object $o \in \mathbb{U}_\Gamma$ as a **datum** simply by viewing it as $o \in \mathbb{S}$. So, we define the denotation of o as a datum by $\Gamma[\![o]\!]_\mathrm{d} := o$. The datum-view and program-view are related by the equality: $\Gamma[\![o]\!]_\mathrm{d} = \Gamma[\!['o]\!]_\mathrm{p}$, which holds for any $o \in \mathbb{U}_\Gamma$.

The denotation of an Object as a **function** is given only when it is a Function (§**??**). Given a realizable Env Γ and a Function $f \in \mathbb{U}_\Gamma$, we can define a meta-level function:

$$\mathsf{f} : \mathbb{S} \dashrightarrow \mathbb{S}$$

by putting $\mathsf{f}(x) = v$ iff $\Gamma \vdash (x) \mapsto_f v \dashv \Gamma'$ for some Γ'. (To simplify the argument, we assume that f is defined by defun with the pattern $p = (x)$.) We write $\Gamma[\![f]\!]_\mathrm{f}$ for f and call it the denotation of f as a function. The function-view and program-view are related by the equality: $\Gamma[\![f]\!]_\mathrm{f}(\Gamma[\![x]\!]_\mathrm{p}) \simeq \Gamma[\![\,(f\ x)\,]\!]_\mathrm{p}$ which holds for any $x \in \mathbb{U}_\Gamma$, where '$\simeq$' means that if one side of the equation has a definite value, the other side also has the same value.

The **class**-view is given only to a Z Class (§**??**). For a Class C, its denotation as a class (written $\Gamma[\![C]\!]_\mathrm{c}$) is the meta-level class that corresponds to C. For example, we have `(<Class>|<Class>)` $\in \langle$Class$\rangle$. The class-view and program-view are related by the equivalence: $\Gamma[\![x]\!]_\mathrm{p} \in \Gamma[\![C]\!]_\mathrm{c} \Leftrightarrow \Gamma[\![\,(\mathtt{in?}\ x\ C)\,]\!]_\mathrm{p} = ()$ $(x \in \mathbb{U}_\Gamma)$.

In summary, among the four views, the program-view is the most basic since the other three views can be characterized by the program-view as we have just seen.

5. Conclusions

We have developed a constructive theory of objects based on the *fundamental principle of object creation*. The fundamental principle we proposed in this paper is inspired by a similar principle proposed by Conway [**?**] to develop a theory of numbers and games. The difference is that we have created our universe based on a constructive view of objects, while Conway created his universe based on a platonistic view without imposing the freeness conditions. An application of our principle to λ-calculus can be found in [**?**].

The development of our open-ended class $\langle$Object$\rangle$ is influenced by Martin-Löf's development of dependent type theories (see, e.g., [**?**, **?**]), but the crucial difference between these developments is that we view any object as an instance of its mother Class while Martin-Löf views any object as an element of a type. We believe that, from a foundational point of view, the notion of *class* is more basic than that of *type* since, in our theory, a Class is always associated with a concept.

We have given a big-step operational semantics of the language Z by defining an evaluation relation on Z Objects. We already gave a big-step semantics of Hyperlisp in [**?**, **?**], but here the rules are more elaborate since Z admits dynamic changes of both global and local Envs. We have also given a denotational semantics of Z, by providing 4 different views of Z Objects.

Our theory of object is *intensional* since our universe must be created following the fundamental principle with the freeness conditions. By taking this view, however, equality of two Functions can be easily decided in Z. We also remark that, at meta-level, we can view Z Functions extensionally by taking the program-view. For example, let Γ be the Env we obtained, in §**??**, after we defined the two factorial Functions f and g where f (g) computes the factorial by recursion (iteration, resp.). Then, we have $\Gamma[\![f]\!]_\mathrm{p} \neq \Gamma[\![g]\!]_\mathrm{p}$, but $\Gamma[\![f]\!]_\mathrm{f} = \Gamma[\![g]\!]_\mathrm{f}$ as a meta-level function on $\langle$Nat$\rangle$. Our theory is natural since we cannot understand the *contents* of mathematics unless, first of all, we understand the *syntax* of the language in which mathematics is written.

Z is a Lisp-like (see, e.g. [**?**]) language in the sense that both data and programs are represented by symbolic objects. The biggest difference between Z and Lisp is that every Z object belongs to its mother Class where the mother Class itself is a Z object, but this is not the case in Lisp since there are no built-in classes in Lisp (although there are some extensions (e.g., CLOS [**?**]) of Lisp with classes as objects. Moreover, each Z Class is created as an Z object by reifying (see Quine [**?**] for a philosophical account of the concept of reification) the corresponding meta-level class listed in Figure 1. In fact, we first analyzed the primitive concepts that are necessary to realize a Turing complete language from scratch, and obtained the data structure of Z by reifying these primitive concepts. Therefore, Figure 1 can been seen as a *concept map* which shows a minimal set of primitive concepts necessary to develop the notion of computation.

The present work is a part of our long-term project ([**?**]) of creating a computer environment for supporting human mathematical *activity*. Philosophical motivation for the project is as follows. As a core subject of science, mathe-

$$
\begin{array}{rr}
\mathsf{nil}: & \to \langle \text{List} \rangle \\
\mathsf{cons}: & \langle \text{Object} \rangle\ \langle \text{List} \rangle \to \langle \text{List} \rangle \\
\mathsf{t}: & \to \langle \text{Tuple} \rangle \\
\mathsf{snoc}: & \langle \text{Object} \rangle\ \langle \text{Tuple} \rangle \to \langle \text{Tuple} \rangle \\
\mathsf{constant}: & \langle \text{Object} \rangle \to \langle \text{Constant} \rangle \\
\mathsf{variable}: & \langle \text{Object} \rangle \to \langle \text{Variable} \rangle \\
\mathsf{class}: & \langle \text{class-name} \rangle \to \langle \text{Class} \rangle \\
\mathsf{zero}: & \to \langle \text{Nat} \rangle \\
\mathsf{succ}: & \langle \text{Nat} \rangle \to \langle \text{Nat} \rangle \\
\mathsf{empty}: & \to \langle \text{Env} \rangle \\
\mathsf{put}: & \langle \text{symbol} \rangle\ \langle \text{Object} \rangle\ \langle \text{Env} \rangle \to \langle \text{Env} \rangle \\
\mathsf{function}: & \langle \text{fun-name} \rangle \to \langle \text{Function} \rangle \\
\mathsf{defun}: & \langle \text{Constant} \rangle\ \langle \text{pattern} \rangle\ \langle \text{Tuple} \rangle \to \langle \text{Function} \rangle \\
\mathsf{defmacro}: & \langle \text{Constant} \rangle\ \langle \text{pattern} \rangle\ \langle \text{Tuple} \rangle \to \langle \text{Function} \rangle
\end{array}
$$

Figure 3. Signature of $\langle \text{Object} \rangle$

matics is not a completed collection of mathematical knowledge. Rather, our mathematical knowledge is dynamically extended, day by day, by our scientific activity. The extensibility, or the open-endedness, of mathematics comes from a key feature of mathematics: introduction of new *concepts* and *objects* by definitions. Thus it is essential for our project to *formalize* and *implement* this key feature. Based on this motivation, we have introduced a theory of objects with the aim of supporting the key feature. The language Z, as defined here, supports introduction of new functions, but does not support introduction of new concepts. We have a plan of adding this feature by incorporating a function, say `defclass`, which enables us to add new classes to $\mathbb{U}$ dynamically. We are also hoping that, by the introduction of this feature, Z will become a class-based functional object-oriented programming language.

A. Specification of $\langle \text{Object} \rangle$

We give the specification of the 14 classes listed in Figure 1. Of these classes 8 are basic classes and they are specified by the signature $\Sigma_{\langle \text{Object} \rangle}$ shown in Figure 3. For example, the second item says that cons is to be a method which creates a new List from an object and an List. However, as we have seen in §**??**, in our implementation, the methods defun and defmacro may be applied only if their first argument, a Constant, satifies a special side-condition. The specification given here does not cover this point.

The 6 derived classes are specified as follows. The classes $\langle \text{Object} \rangle$, $\langle \text{symbol} \rangle$ and $\langle \text{atom} \rangle$ are specified by the following equations:

$$
\begin{aligned}
\langle \text{Object} \rangle &:= \langle \text{List} \rangle + \langle \text{Tuple} \rangle + \langle \text{symbol} \rangle + \langle \text{atom} \rangle \\
\langle \text{symbol} \rangle &:= \langle \text{Constant} \rangle + \langle \text{Variable} \rangle \\
\langle \text{atom} \rangle &:= \langle \text{Nat} \rangle + \langle \text{Env} \rangle + \langle \text{Function} \rangle + \langle \text{Class} \rangle
\end{aligned}
$$

We specify the remaining derived classes $\langle \text{pattern} \rangle$, $\langle \text{class-name} \rangle$ and $\langle \text{fun-name} \rangle$ in terms of axioms for the corresponding concepts pattern, class-name and fun-name. First, we introduce constants $\mathsf{c1}, \ldots, \mathsf{c14}$ for the 14 class-names together with the 15 axioms:

$$\text{class-name}(\mathsf{c1}),\ \ldots\ ,\ \text{class-name}(\mathsf{c14})$$

and

$$\text{Constant}(x) \Leftarrow \text{class-name}(x)$$

The first 14 axioms assert that each of the 14 constants is a class-name, and the last axiom asserts that any class-name is a Constant. We can specify the concept fun-name similarly. To specify the concept pattern, we introduce a constant r, and we also specify the concept decl (declaration). In the axioms for decl, we we write $(c\ x)$ for $\mathsf{snoc}(c, \mathsf{snoc}(x, \mathsf{t}()))$ and understand $(\mathsf{r}\ c\ x)$ similarly.

$$
\begin{aligned}
\text{Constant}(\mathsf{r}) &\Leftarrow \\
\text{decl}(x) &\Leftarrow \text{Variable}(x) \\
\text{decl}((c\ x)) &\Leftarrow \text{class-name}(c),\ \text{Variable}(x) \\
\text{decl}((\mathsf{r}\ c\ x)) &\Leftarrow \text{class-name}(c),\ \text{Variable}(x) \\
\text{pattern}(\mathsf{t}()) &\Leftarrow \\
\text{pattern}(\mathsf{snoc}(d, p)) &\Leftarrow \text{decl}(d),\ \text{pattern}(p)
\end{aligned}
$$

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*, 2nd edition. The MIT Press, 1996.

[2] J. H. Conway. *On Numbers and Games*, 2nd edition. A K Peters Ltd., 2001.

[3] G. Frege. On Function and Concept. in *The Frege Reader*, M. Beaney (ed.), Blackwell Publishing, 130–148, 1997.

[4] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[5] B. Nordström, K. Peterson and J. M. Smith. *Programming in Martin-Löf's Type Theory – An Introduction –*, Clarendon Press, Oxford, 1990.

[6] M. Sato and M. Hagiya. Hyperlisp. *Proceedings of the International Symposium on Algorithmic Language*, 251–269, North-Holland, 1981.

[7] M. Sato. Theory of symbolic expressions, I. *Theoretical Computer Science*, 22:19–55, 1983.

[8] M. Sato and R. Pollack. External and internal syntax of the λ-calculus, *Journal of Symbolic Computation*, to appear.

[9] M. Sato. A framework for checking proofs naturally, *Journal of Intelligent Information Systems*, 31:111–125, 2008.

[10] S. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, 1988.

[11] W. V. O. Quine. *From a Logical Point of View* (2nd ed., revised) Harvard University Press, Cambridge, MA, 1980.