Primitive Recursion on Higher Types *

Stanislaw Ambroszkiewicz
Institute of Computer Science, Polish Academy of Sciences al. Jana Kazimierza 5,
PL-01-248 Warsaw, Poland
e-mail: sambrosz@ipipan.waw.pl,
www.ipipan.waw.pl/mas/stan/

ABSTRACT

A revision of the basic concepts of type, function (called here operation), and relation is proposed. A simple generic method is presented for constructing operations and types as concrete finite structures parameterized by natural numbers. The method gives rise to build inductively so called Universe intended to contain all what can be *effectively constructed*. For any (higher order) type A the operation of type
$Rec_A : (N; (N \to (A \to A))) \to (A \to A)$ is constructed that corresponds to primitive recursion of Grzegorzyk [8] and Girard [4].

Keywords

Higher order types, primitive recursion

1. INTRODUCTION

It is a continuation of the work of Professor Andrzej Grzegorczyk [8] (who was inspired by the System T of Kurt Gödel [5]) concerning recursive objects of all finite types.

The phrase *effectively constructed objects* may be seen as a generalization of the notion of *recursive objects*. Objects can be represented as finite (usually parameterized) structures. Universe is understood here as a collection of all generic constructible objects.

In the Universe, constructability is understood literally, i.e., it is not definability, like general recursive functions (according to Gödel-Herbrand) that are defined by equations in Peano Arithmetic along with proofs that the functions are global, that is, defined for all their arguments. Objects are not regarded as terms in lambda calculus or in combinatory logic.

Most theories formalizing the notion of effective constructability (earlier it was computability) are based on the lambda abstraction introduced by Alonzo Church that in principle was to capture the notion of function and computation. Having a term with a free variable, it is easy to make it a function by applying lambda operator. Unlimited application of lambda abstraction results in contradiction (is meaningless), i.e., some terms cannot be reduced to the normal form. This very reduction is regarded as computation. Introduction of types and restricting lambda abstraction only to typed variables results in a very simple type theory.

Inspired by System T, Jean-Yves Girard created system F [4], [3]; independently also by John C. Reynolds [14]. Since System F uses lambda and Lambda abstraction (variables run over types as objects), the terms are not explicit constructions. System F is very smart in its form, however, it is still a formal theory with term reduction as computation; it has strong normalization property.

Per Martin-Löf Type Theory (ML TT for short) [13] was intended to be an alternative foundation of Mathematics based on constructivism asserting that to construct a mathematical object is the same as to prove that it exists. This is very close to the Curry-Howard correspondence *propositions as types*. In ML TT, there are types for equality, and a cumulative hierarchy of universes. However, ML TT is a formal theory, and it uses lambda abstraction. Searching for a grounding (concrete semantics) for ML TT by the Author long time ago, was the primary inspiration for the Universe presented in this work.

ML TT, and System F are based on lambda and Lambda abstraction, so that in their syntactic form they correspond to the term rewriting systems.

In this work lambda and Lambda abstractions are challenged. It is an attempt to show that the same (as in System F), and perhaps more, can be achieved by explicit and concrete constructions, even though these constructions are not so concise and smart as the corresponding terms in System T. The proposed method relates rather to the approach where explicit constructors are used. In this sense, it continues the idea of Grzegorczyk's combinators [8], and in some sense also combinators in Haskell B. Curry [2] combinatory logic.

Effective construction of an object cannot use actual infinity. If it is an inductive construction, then the induction parameter must be shown explicitly in the construction. For any fixed value of the parameter the construction must be a finite structure. The Universe presented in this paper is supposed to consist only of such objects. Objects are not identified with terms whereas computations are not term rewritings. Although, in

*The work was supported by the grants:
RobREx - Autonomia dla robotów ratowniczo-eksploracyjnych. Grant NCBR Nr PBS1/A3/8/2012 w ramach Programu Badań Stosowanych w Obszarze Technologie informacyjne, elektronika, automatyka i robotyka, w Ścieżce A.
oraz *IT SOA - Nowe technologie informacyjne dla elektronicznej gospodarki i Społeczeństwa informacyjnego oparte na paradygmacie SOA*; Program Operacyjny Innowacyjna Gospodarka: Działanie 1.3.1..

computations all can be reduced to the primitive types, higher order types and their objects correspond in programming to sophisticated data structures and their instances.

The proposed Universe is not yet another formal theory of types. It is intended to be a grounding for some formal theories as well as a generic method for constructing objects corresponding to data structures in programming.

Universe is strongly related to computable functionals (Stephen C. Kleene [9][10][11], Georg Kreisler [12], Grzegorczyk [6] and [7], as well as to Richard Platek & Dana Scott PCF^{++} [15, 16]).

2. FOUNDATIONS

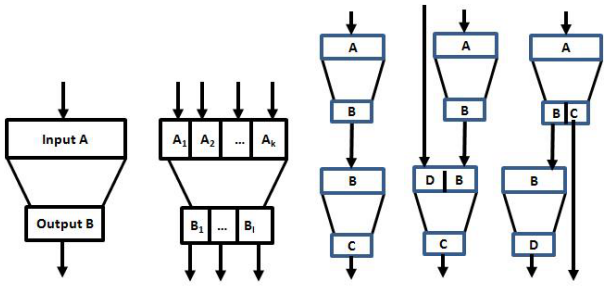

Figure 1: On the left, graphical schema of operation; on the right, composition of two operations.

This section and the next one may be seen as naive because there are no technicalities here. However, the aim is to present the most primitive notions as simple as possible. These notions (as basic elements for constructions) are *types*, *objects*, and *operations* that process input objects into output objects. Operation is a synonym for function.

Object a of type A is denoted by $a : A$. The type of operation is determined by the type of its input (say A) and the type of its output (say B), is denoted by $A \to B$.

Primitive type may be interpreted as data link (communications channel) whereas object of that type as a signal transmitted in this channel. The interpretation may be extended for complex types.

Simple operation has one input and one output, however, in general, operation may have multiple inputs as well as multiple outputs, see Fig. 1. The type of operation g having multiple inputs and multiple outputs is denoted by $g : (A_1; A_2; ...; A_k) \to (B_1; B_2; ...; B_l)$

Operation $f : A \to B$ may be applied to its argument, i.e., input object $a : A$. The output of this application is denoted by $f(a)$. For operations with multiple input, application may be partial, i.e., only for some of the inputs (say a_i and a_j). Then it is denoted by $g(a_j; a_i; *)$. Application is amorphous, however, if the type of the operation and the types of arguments are fixed, then application may be considered as an operation.

There are no variables in our approach. In lambda calculus variables serve to denote inputs. In combinatory logic any combinator has exactly one input. Operation having many inputs can be (equivalently in some sense) transformed (by *currying*) into operation having one input. It will turn out in Section **??** that currying is an operation.

Composition of two operations consists in joining an output of one operation to an input of another operation. The type of the output and the type of the input must be the same, see Fig. 1. Composition is amorphous, however, if the operation types are fixed, then composition may be considered as an operation.

The Universe will be developed inductively (actually, by transfinite induction) starting with primitive constructors, destructors and primitive types. At each inductive level, new primitives will be added. The primitives are natural consequences of the constructions methods from the previous levels and give rise to new methods. Each level is potentially infinite. The Universe is never ending story. Once construction methods are completed for one level, it gives rise to the construction of the next level and new methods. There are always next levels that contribute essential and qualitatively new constructions to the Universe.

This constitutes a bit intuitive and informal foundation for the Universe. In the next sections the idea is developed fully and precisely.

3. LEVEL ZERO

Level 0 of the Universe consists of primitive constructors of types, primitive types, and related primitive operations. On the level 1, the types forming level 0 will be treated as objects, analogously for higher levels. The levels of the Universe correspond to an infinite well-founded typing hierarchy of sorts in CoIC [1].

3.1 Type constructors

Keeping in mind the interpretation of types as telecommunication links, there are three basic type constructors. Let A and B denote types.

- $\times$ product of two types $A \times B$; as one double link consisting of A and B. Signals (objects) are transmitted in $A \times B$ simultaneously.
- $+$ disjoin union $A + B$; two links are joined into one single link. Signal (object) transmitted in this link is preceded by a label indicating the type of this object.
- $\to$ arrow, operations type $A \to B$; A is input type, whereas B is output type.

These three basic constructors are independent of primitive types. On the level 1 these constructors will be considered as operations on types, and new type constructors will be introduced.

Product and disjoint union are natural and their nesting, like $(A \times B) + (C \times D)$, has obvious interpretation. The meaning of operation type is a bit more harder to grasp, especially if input is again an operation type, like $(A \to B) \to C$. The problem is how to interpret operation as an object. Actually this problem can be reduced to grasping properly what input types and output types

are in an operation. Fig. 2 may help. Input is interpreted as socket board where each socket corresponds to a link that is a single type. The same for the output.

Type of operation is again a socket board consisting of two parts. The upper part is a socket board as the input type. The bottom part is a socket board as the output type, see Fig. 3. Putting an object $a : A$ into a socket

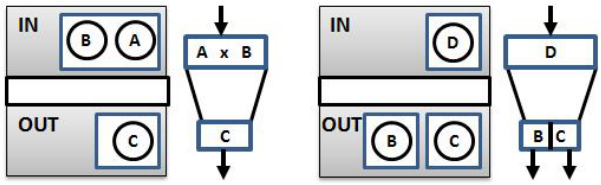

Figure 2: Another pictorial presentation of operations: on the left, there is operation with input socket board consisting of $A \times B$ and one output board consisting of type C; on the right, output socket board consists of two independent types B and C.

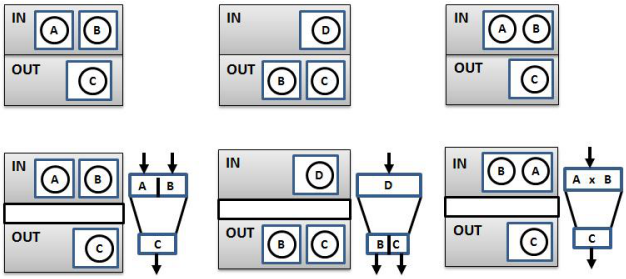

Figure 3: Operations and their types: operations are presented in the bottom row whereas their types in the top row.

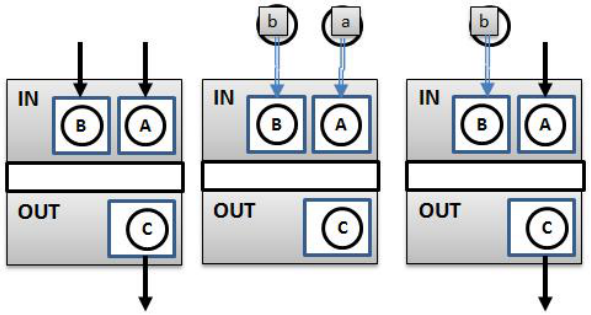

Figure 4. An operation applied to objects.

of type A is interpreted as a transmission of the object via this socket. For an operation $f : A \to C$ putting an object $a : A$ into the input socket of type A means the application $f(a)$. So that the object $f(a)$ will appear at the output socket of type B.

For operation $f : (B; A) \to C$ the application $f(b, a) : C$ is just an object at the output socket C, see Fig. 4. However, $f(b, *)$ is still an operation of type $A \to C$.

Once this is clear, it is also easy to grasp what it means to apply operation $F : (A \to B) \to C$ to an input object $h : A \to B$. The input socket board of the operation F is of type $A \to B$. Putting object h into the input socket board of F means connecting all the sockets of input and output of h to the input board of the operation F, see Fig. 5, where input socket of h (of type A) is connected to the input socket (of type A) of the input board of F, whereas output socket of h (of type B) is connected to the output socket (of type B) of the input board of operation F. Once it is done, the result of the application $F(h)$ is at the socket C of the output board of the operation F.

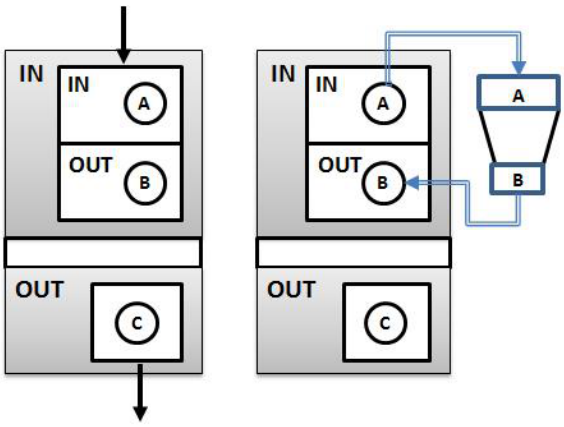

Figure 5: Application of operation $F : (A \to B) \to C$ to object $h : A \to B$.

This is the crucial point of our approach. Types are interpreted as sockets whereas input types and output types as socket boards. Operation type is interpreted as a complex board consisting of input board and output board. This gives rise to interpret types as objects at the level 1 of the Universe.

Constructors of objects corresponding to product, disjoin union, and arrow (operation type) are as follows. Let $a : A$ and $b : B$.

- for product: $join_{A,B}$ is an operations of type $(A; B) \to (A \times B)$ such that $join_{A,B}(a; b)$ is an object of type $A \times B$ denoted as a pair by (a, b).
- for disjoin union: $plus^A_{A,B} : A \to (A + B)$ and $plus^B_{A,B} : B \to (A + B)$. For $a : A$ and $b : B$, $plus^A_{A,B}(a)$ and $plus^B_{A,B}(b)$ are objects of type $A + B$.
- for arrow: for any $a : A$ there is the constant operation of type $B \to A$, such that for any $b : B$, it returns a as its output. More generally, $const_{A,B} : A \to (B \to A)$, such that operation $const_{A,B}(a) : B \to A$ returns always a as its output.

It is important that the equality symbol "=" in not used even for description.

Destructors for product, disjoin union, and arrow.

- $proj_{A,B} : (A \times B) \to (A; B)$. For any (a, b) of type $A \times B$, projection returns two output objects denoted by $proj^A_{A,B}((a, b)) : A$ and $proj^B_{A,B}((a, b)) :$

B. Composition of $join_{A,B}$ and $(proj^A_{A,B}; proj^B_{A,B})$ gives two identity operations: $id_A : A \to A$ and $id_B : B \to B$, that return the input object as the output. Although identity operation (say for type A) may be identified with a link of type A, it is useful in constructions.

- $get_{A,B} : (A + B) \to (A; B)$. For an input object of type $A + B$, it returns only one output: either $get^A_{A,B}$ or $get^B_{A,B}$. Although this operation has two outputs, once it is applied, only one output has object; it is determined by the input object.

- $apply_{A\to B,A}$. Application as operation indexed by types A i B is of type $((A \to B); A) \to B$. For any input objects $f : A \to B$ and $a : A$, it returns as the output $apply_{A\to B,A}(f; a)$, i.e., the same as the amorphous application $f(a)$ (being also a destructor for arrow type). Applications as operations are used in constructions.

Operation $apply_{A\to B,A} : ((A \to B); A) \to B$ is interpreted as a specific board consisting of linked sockets, see Fig. 6. Generally, operation *apply* may be more

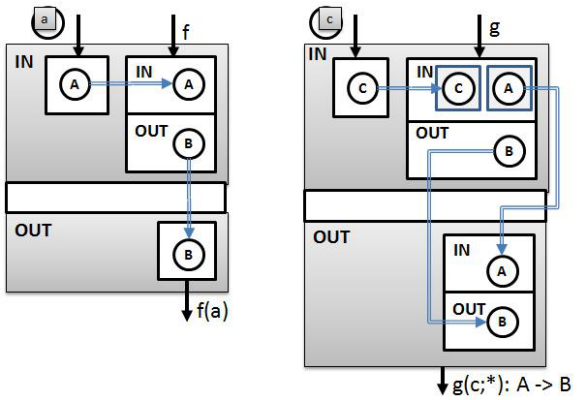

Figure 6: Applications as operations: simple $apply_{A\to B,A} : ((A \to B); A) \to B$**, and complex** $apply_{(((C;A)\to B),C)}$

complex, i.e., may have multiple inputs for example, it maybe of type $(((C; A) \to B), C) \to (A \to B)$, see Fig. 6.

4. PRELIMINARY CONCLUSION

Since there is no more space here to present the complete work, to continue the reading please go to the paper *Types and operations* `http://arxiv.org/abs/1501.03043`.

REFERENCES

[1] T. Coquand. Coq proof assistant. chapter 4 calculus of inductive constructions. www http://coq.inria.fr/doc/Reference-Manual006.html, 2014. Site on www.

[2] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. Amsterdam: North Holland, 1958.

[3] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7. Cambridge University Press (Cambridge Tracts in Theoretical Computer Science), 1990.

[4] J.Y. Girard. Une extension de l'interpretation de godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et dans la theorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium 1971*.

[5] K. Gödel. Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *Dialectica*, 10:280 – 287, 1958.

[6] A. Grzegorczyk. Computable functionals. *Fundamenta Mathematicae*, 42:168–202, 1955.

[7] A. Grzegorczyk. On the definition of computable functionals. *Fundamenta Mathematicae*, 42:232–239, 1955.

[8] A. Grzegorczyk. Recursive objects in all finite types. *Fundamenta Mathematicae*, 54:73–93, 1964.

[9] S. C. Kleene. Countable functionals. *Constructivity in Mathematics: Proceedings of the colloquium held at Amsterdam*, pages 81–100, 1959.

[10] S. C. Kleene. Recursive functionals and quantifiers of finite types i. *Transactions of the American Mathematical Society*, 91:1–52, 1959.

[11] S. C. Kleene. Recursive functionals and quantifiers of finite types ii. *Transactions of the American Mathematical Society*, 108:106–142, 1963.

[12] G. Kreisel. Interpretation of analysis by means of functionals of finite type,. *Constructivity in Mathematics: Proceedings of the colloquium held at Amsterdam*, pages 101–128, 1959.

[13] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloqium 1973*.

[14] J. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation 1974*, pages 408–425. Paris, France, 1974.

[15] D. S. Scott. A theory of computable functions of higher type. *University of Oxford*, 1969.

[16] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993.