The Language FLAC: Computational Model and Modularity

Victor L. Kistlerov

Institute of Information Transmission Problems
Russian Academy of Sciences
kistler@iitp.ru

Abstract. The development of a language for algebraic computation actually needs a specific computation model that supports typical behavior of formula manipulation. For this purpose an "intentional" model with suspended computations was developed, and language FLAC with specific modularity is an implementation of the model.

The development of a language for algebraic computation actually needs a specific computational model that supports typical essential features of algebraic computation.

The base of the model is the *concept of suspended computations*, that regards all functions as partial ones, and result of computation of $f(x_0)$, that is a call of function f at point x_0, specifically depends on the domain of f. It is fundamentally, that if $x_0 \notin Dom(f)$ then the function call $f(x_0)$, in spite of conventional approach, is suspended, i.e. function f is extended at the point x_0 by so called "intensional", that is a ground term $f(x_0)$.The intensional is appended to the set of values, and the computation continues on the extended set of values.

A reason for the model is substantiated by examples of regular means for imitation of constructors for numbers and algebraic expressions. For example, number -1 is regarded as an intesional, arisen as a suspension of function `subtract` for naturals in expression `subtract(0,1)`. Similarly $1/2$ is the same for natural function `divide`: `divide(1,2)`; the imaginary number I is intensinal, arisen for real function `sqrt` at the point -1. And finaly, the polynomial $x + 1$ is the intensional of the function `add`.

The FLAC language [1] (is an abbreviation for Functional Language for Algebraic Computation) is a functional language much similar to Refal [2] with conventional elements of the languages like terms, variables, pattern matching, alternation and recursion. Any function defined in a program is regarded as partial one, and programming in FLAC is extending the functions.

Here is a syntax of a simple version of the language.

```
      Program  =  Sentence   |   {Sentence";"}

Snt:  Sentence  = Term  "="  Expression

T:    Term   =   Simple-Term   |   Compound-Term
St:     Simple-Term   =   Simple-Ground-Term   |   Variable
SGt:    Simple-Ground-Term = Identifier  |  Number  |  Literal

Ct:   Compound-Term   =   Head  "("  List  ")"
H:      Head   =   Name   |   Term-Variable
Name:   Name   =   Identifier
L:    List   =   Term   |   Term  {","  List}

Gt:  Ground-Term  =  Simple-Ground-Term | Compound-Ground-Term
CGt:    Compound-Ground-Term   =  Name  "(" Ground-Term-List ")"
Gl:     Ground-Term-List   =  Ground-Term {"," Ground-Term-List}


V:   Variable   =   Term-Variable   |   List-Variable
Vt:    Term-Variable   =   "&" Identifier
Vl:    List-Variable   =   "#" Identifier

Id:      Identifier
Num:     Number
Liter:   Literal
```

The following is the famous factorial function written in FLAC:

```
fac(0) = 1;
fac(&n) = &n * fac(&n-1);
```

The most specific feature of the language is to support suspended computations. Any ground term is regarded as functional call and is trying to be converted. If a ground term t calls a function f outside its domain then the term t is suspended and converted to a ground term t' which denotes the result of suspension of the term t. For the sake of usability the t' is represented literally by the ground term t itself. As a result of the suspension the term t' is appending to the set of values for further computations. Actually, we have to consider every resulted ground term as an intensional of suspended computation.

Program is a sequence of definitions. The sentences of a function description give alternative patterns, that are tried one by one. To convert a term `f(a)` pattern matching process starts with the first sentence of description of the function `f`. Each Term-Variable can take only one Ground-Term, and List-Variable can take Ground-Term-List, when matching from left to right. If it is impossible to

match the current sentence, the process restarts from the beginning of the next sentence.

For example, let function `apply` applies first argument to each element of the list of the second argument, that is the compound ground term with the name `Terms`:

```
apply(&f,Terms(&x,#l))  =   &f(&x), apply(&f,Terms(#l));
apply(&f,Terms(#l))     =   #l;
```

Compound ground terms are also used for data type representation: the name of a compound term is used as the name of type. For example, matrix

$$A = \begin{pmatrix} cos(\varphi) & -sin(\varphi) \\ sin(\varphi) & cos(\varphi) \end{pmatrix}$$

may be represented as:

```
A = Mat(2,2, Row(cos(fi), -sin(fi)), Row(sin(fi), cos(fi)));
```

Each sentence of description of function f defines also the domain D_i^f, and so the function has the domain $D^f = \bigcup_i d_i^f$.

Usually, programming systems have modularity. Conventional tradition allows make definition of one function only in one of composed modules. However, it is not convenient for algebraic computations, because mathematical tradition dictates necessity of extension of earlier defined operations for new mathematical objects.

FLAC has modularity that allows a once defined function to be extended in other modules. So, if function f has definitions in modules $m_1, \ldots, m_n$ that are composed into a single-module M, then the domain of the function f is $D_m^f = \bigcup_k D_{m_k}^f$.

For instance, if functions `+` and `*` were before only numeric ones, and we have made new module with defined matrix operations named `addmat` and `mulmat`, then we can just add extra definitions:

```
Mat(#l1) + Mat(#l2)  =  addmat(Mat(#l1), Mat(#l2));
Mat(#l1) * Mat(#l2)  =  mulmat(Mat(#l1), Mat(#l2));
```

and now use it in algebraic manner:

$$B = A * A + M;$$

Moreover, on modules $m_1, \ldots, m_n$ may be defined a partial order with corresponding semantics of the function f on each branch of the tree.

Though resulted domain of f does not depend on order of modules in M, but result of computation may essentially depend on it.

References

1. V. Kistlerov. The principals of development of the Computer Algebra Language. Preprint of Institute of Control Sciences, Moscow, 1987. (In Russian).
2. V. Turchin Refal-5: Programming Guide and Reference Manual, New England Publishing Co. Holyoke MA, 1989.
3. Victor Kistlerov. An operational semantics of suspended computations. In *Complex Systems: Control And Modeling Problems*, Samara, June 1999.