INVERSION and METACOMPUTATION

Alexander Romanenko*
Institute for the Physical Chemistry
Academy of Sciences of the USSR
Leninskyi prosp.31,
SU-117915, Moscow, USSR

ABSTRACT

The problems of constructing an inverse function definition from definition of a direct one are considered. The suggested approach is based on ideas of metacomputation in the applicative programming language Refal. The proposed extensions enable to express the inverse semantics on the ground language level. Thus the new language, Refal-R becomes a functional logic language. An inverter is used as a tool to obtain inverse programs. The supercompiler - a metaevaluator over Refal - is being extended to use the inverter and to manage Refal-R.

1. INTRODUCTION

Main goals of this report are: to outline some approaches to automatic transformation between programs for a direct and an inverse function; to develop a language with a built-in ability to represent results of inverting; to suggest an integrated programming system, based on metacomputation with inversion as a natural and essential constituent part.

1.1. Motivation

The Inverse Problem is treated quite differently by Mathematician and Programmer. Inverse functions are mentioned only twice in the textbook on recursive functions [Cutland 79], there are works on the complexity of inverse functions - and that is almost all for a Mathematician to deal with this problem. But Programmer only starts at this point, trying to find efficient procedures for function inversion though they may not have a rigorous mathematical inference. The thinking style of the author is closer to the latter.

"The straightforward approach would be to try to develop a formalized theory of Turing machines in which length of a computation is defined and then to try to get a decision procedure for this theory. This is known to be a hopeless task. Systems much simpler than Turing machine theory have been shown to have unsolvable decision procedures. So, we look for a way of evading these difficulties." [McCarthy 56]

Logic programming can be considered as an approach to inverse problem solving. Given a goal or result the instantiantion of input variables are being searched to meet the goal. The *directional transparency* of logic programs may be considered as their advantage because they are easier to read or understand [Reddy 84]. But *forced* directional transparency resulting in inability to express properly functional directionality can be treated as shortage of logic languages [Reddy 86]. It becomes even more evident when a program is intended to be an object for manipulation. As far as I know, only cut-free subsets of Prolog were considered as objects for partial evaluation [Fuller,Abramsky 88; Kursawe 88].

Metacomputation was introduced and successfully developed in the frame of functional programming [Futamura 71; Turchin 72, 86; Jones 88]. It is a computation where programs are treated as data objects. Partial evaluation is a special case of metacomputation. Metacomputation or supercompilation includes partial evaluation but cannot be reduced to it; it is a much wider framework for general function transformations. Function inversion is one of such transformations. The main tool here is the supercompiler - a program transformer of a certain type. The way it works is the following. It builds a model of a program in some form, runs the program with unknown values of variables (it is called *driving*), and try to make this model finite and self-sufficient with the use of *generalization* [Turchin 86,88].

The idea to combine logical and functional features in one programming language is not new. But we are trying to do such a combination in the frame of metacomputation, when the every program in the suggested language is considered as an object for further transformation and not as a final product. The supercompiler paradigm allows us to revise some old conceptions about optimal ways of solving problems, in particularly, a breadth-first search in the computation tree. In the paper [Shoman,McDermott 84] such an approach was called "an obvious non-solution to all inversion problems" with "theoretical completeness and impracticality". We hope that

*This research is partially supported by scientific grant of JV 'DIALOGUE'

with the supercompiler we can use its theoretical completeness and overcome its impracticality.

1.2. History and related works

Perhaps the first work addressing the problem of inverting recursive functions was the paper by J.McCarthy [McCarthy 56], where he discusses the possible improvements of the general enumeration procedure.

In the context of Refal the first results in this area were obtained in 1972 [Turchin 72; Glueck,Turchin 90]. The formulation of the inverse problem was as follows. Let *Equal(x,y)* be an equality predicate and *F*(*x*) a function. Find such an x that *Equal(F(x)*,C) is TRUE, where *x* is a variable and C is a constant. For every C the value of *x* can be found through the driving algorithm [Turchin 72]. For a given Refal program it builds an evaluation tree (generally infinite) for this program. Let *Drive* be the driving algorithm for Refal. When applied to *Equal(F*(x),C), *Drive* tabulates it in the following way:

$Equal(F(A_1),C) = \text{TRUE}$
$Equal(F(A_2),C) = \text{TRUE}$
....

where A_1, A_2,.. are some constants or pattern expressions (i.e. ones including variables). In general this process is infinite but sometimes it may terminate, the last generated path being:

$Equal(F(x),C) = \text{FALSE}$

Let *Extract* be a function that extracts arguments of the function *F* (answer) from a result of *Drive*. Then a computation of the expression:

Extract(*Drive*('Equal(F(x),C)'))

yields sequentially A_1, A_2 etc. This algorithm was called *Universal Resolving Algorithm* (URA) [Turchin 72]. In Prolog-like terminology, this is a breadth-first search in the computation tree.

It can be shown that inverse function building can as well be carried out with the help of driving. Constant C must be replaced by a variable *y* and then the expression

Extract(*Drive*('Equal(F(x),y)'))

is specialized with respect to variable *y*. So URA can be treated as an interpreter defining the inverse semantics of a programming language on a meta-level. S.M.Abramov has considered Refal as the logic language with URA instead of the standard Refal-interepreter. Using URA and Specializer a set of metalinguistic formulae can be obtained [Abramov 88,89]. We are interested in the following:

Inverse program
    P = *Spec*(*Ura*,'SD',*P*)
Program inverter
    Inv = *Spec*(*Spec*,'SSD',*Ura*,'SD')
Inverter generator
    GeInv = *Spec*(*Spec*,'SSDS',*Spec*,'SSD','SD')

The *Spec* (specializer) is a program *Spec*(*p*,*M*,*x*) where *p* is a n-ary program, *M* is a string (mask) with the length n, consisting of characters 'S' (static) and 'D' (dynamic) and showing the necessity to specialize on the *i*-th argument, *x* be a vector of values of static arguments. There is a direct analogy between these formulae and Futamura's projections [Futamura 72; Jones 88]. So, the breadth-first URA provides the completeness and the Specializer or Supercompiler may produce the required efficiency. But the needed power of a specializer must be rather high: only its double self-application yields interesting results.

The supercompiler does not require splitting arguments into static and dynamic parts. Its main procedure is a cleaning-up impossible computation branches. One of our intentions is extending the supercompiler to manage programs in Refal-R. Some of the methods are similar to that ones used in searching functional dependencies in logic programs [Reddy 84; Debray,Warren 89].

The mechanism of *narrowing* was originally introduced in applicative programming [Turchin 72,74,80,86] and classical logic [Slagle 74] almost at the same time. Turchin uses the other naming convention: *contractions* stands for *narrowing substitutions*, *driving* - for the narrowing itself, *Generalized Matching Algorithm* (GMA) stands for unification. The notions were independently reinvented after the appearance of logic programming (see [Reddy 86,87] for references). Refal uses pattern matching as operational semantics, but narrowing and unification (driving and GMA) are embedded into the Supercompiler - a metaevaluator over Refal [Turchin 86]. We lower narrowing from the meta-level down to Refal-R.

Function inversion is also considered in [Harrison 88], where the author describes the general one-argument function inversion algorithm and its specialization in the Backus's FP language. His approach is denotational in its essence, however in the present paper a constructive, operational world is preferred.

In a previous work [Romanenko 88] a task of a program extracting for an inverse function from a program for a direct one had also been regarded. The programs were written in Refal. With configurational analysis [Turchin 80] efficient inverse functions were built even in such nontrivial cases as multiplication and the Ackermann's function. However, the general case of inversion was not considered. Now we try to fill this gap.

1.3. Outline

The approach suggested in this paper is also based on a supercompiler paradigm but in a different from URA way. We propose a language, Refal-R, in which inverse semantics can be naturally expressed on the ground level. The main extentions of new Refal-R with respect to Refal are set-valued functions and the use of driving or narrowing instead of pattern-matching as the operational semantics of the language. As a result it becomes a *functional logic language* [Reddy 87].

If function *f* is computable and injective then the inverse one is computable [Cutland 80] and the following relation holds: $f^{1}(f(x)) = x$ if x is in *f*'s domain. But if we consider a more general case of multi-valued functions, the above relation may be generalized: $f^{1}(f(X)) \supset X$, where X is a set-variable. The injectivity of *f* - and, hence, the existence of a single-valued f^{1} is an undecidable question. But we can construct a set-valued *approximation* for f^{1} and use the supercompiler to make this approximation as close as possible.

An inverter is used as a tool to obtain inverse programs. It turned out that a simple textual inverter produces correct inverse programs or their approximations. The method is rather general but - as a natural consequence of an exhaustive search explosion - may produce inverted functions with a poor performance. One of the ideas behind this augmentation is the more clear relation between syntactical *reversion* and semantic *inversion* of a program. D.Gris used also this method in [Gris 81] applying it to single-valued, injective functions.

The language is described in the next section. The third section deals with an inversion procedure for the language constructed. Both cases of inversion are considered: full inversion and inversion with respect to any subset of function parameters. The fourth section discusses the possible supercompiler extensions to treat and use the new version of Refal:

- It is possible to suggest a more efficient inversion algorithm. It is based on inverting the computation histories of functions. The resulting inverse function may be comparable by performance with the direct one.
- Extracting a single-valued function from its set-valued prototype, when it is possible or making an approximation closer - is another task for the supercompiler, managing programs in Refal-R.
- *Inverse driving* is a natural mirror for direct driving in the case of a symmetrical invertible language discussed in the conclusion of the fourth section.

2. THE LANGUAGE

The language Refal has a long history and a few dialects. We describe here a generalized version of it with modifications which meet the requirements of inversion. We call the resulting language Refal-R, where the last "R" stands for "Reversible". The reference to 'Refal' will be made in the description of the inherited features and to 'Refal-R' when discussing the new ones.

2.1. Syntax

pgm	:	Programs
fd	:	Function definitions
exp	:	Expressions
fn	:	Function names
vn	:	Variable names
p	:	Paths
c	:	Constructs
cc	:	Complex constructs
cl	:	Clashes
t	:	Terms
S	:	Symbols

```
pgm      ::= fd1;...;fdn
fd       ::= <fn exp1> = exp2 cc
cc       ::= {p1;...;pn} | [p1;...;pn]
p        ::= c1,...,cn
c        ::= cc |cl | #cc
cl       ::= exp1 : exp2
exp      ::= [] | (exp) | vn |
                 <fn exp> | S | exp1 exp2
t        ::= S | (exp)
```

Three types of variables are acceptable in Refal, with the names, beginning with a letter corresponding to the type of the variable: *s-variables*, standing for symbol values, *t-variables* and *e-variables*, standing for terms and expressions respectively. Note that parentheses are not symbols and cannot serve as values of variables.

An expression is *passive* if it does not include activation brackets, it is *active* otherwise. An expression without free variables is a *ground* expression; otherwise it is a *nonground* expression. A *pattern* is a passive, and generally nonground, expression. A passive ground expression is an *object* expression.

An *L-expression* is a pattern which:

- contains no more than one occurrence of every *e*-variable or *t*-variable (such a property is called *linearity* [Reddy 86]);

- contains no more than one e-variable on the top level in every subexpression (i.e., none of its subexpression can be represented as $E_1 e_x E_2 e_y E_3$, where E_1, E_2, E_3 are expressions).

Note that linearity must hold only with respect to *e* or *t*-variables but not for *s*-variables.

Both exp_1 and exp_2 in the function definition (they correspond to its *input* and *output patterns*) and both exp_1 and exp_2 in the clash definition above must be *L*-expressions. A clash may contain function calls in its left side. Its left side and all function arguments must reduce to *L*-expressions after every function call in it is replaced by the output pattern of the corresponding function definition.

2.2. Semantics

The domain of Refal program consists of expressions, Concatenation is a built-in operation and has no special operation sign. As a result Refal program can manage an expression from either of its ends. The domain of Refal-R program consists of *sets* of expressions. Despite we augmented the language with the set datatype, a set itself cannot be treated directly. The function mapping over set can be expressed as: $f(S) = \{f(x) \mid x \in S\}$. So the language preserves good features, for example, referential transparency, etc.[Hughes,O'Donnel 89]

A program consists of ordered sequence of function definitions. The main Refal operation is *pattern matching.*

Given an object expression *E* and an *L*-expression *L*, matching *E*:*L* is defined as finding such a substitution *S* for the variables in *L* that applying *S* to *L* yields *E*. The values assigned to *s*-variables in substitutions must be single symbols, *t*-variables and *e*-variables can take correspondingly any terms and expressions as their values. If there is such a substitution, we say that the matching succeeds, and *E* is recognized as (a special case of) *L*, it fails otherwise.

Consider the more general case of clash *R*:*L*, where *R* may contain variables and function calls. Two possible interpretations of this clash are interesting: matching and unification. The usual Refal semantics is based on matching. Before the matching can be executed *R* must evaluate to an object expression, but for unification it may remain a pattern. For a proper representation of function inversion we need to augment Refal with set-valued functions. This, in turn, leads to the appearance of free variables in the input expression of functions and clashes. The semantics of clashes and parameter passing in Refal-R is based on *narrowing.*

Given two *L*-expressions *R* and *L*, the unification *R*:*L* is defined as finding such a complete set of substitutions $\{ S_i \}$ for the variables in *R* that applying S_i to *R* allows to match the result against *L*. If this set is not empty, we say that the unification succeeds, otherwise the unification fails. Even in more general situation, when *R* is not *L*-expression, it can be shown that there exists only a finite set of possible narrowing substitutions. We shall distinguish particular types of clashes *R*:*L* - *contractions* and *assignments*. A contraction is a clash where *R* is a single variable. Assignment (passive) is a clash where R does not contain activation brackets <>, L consists of a single new variable and its type allows the matching operation to be executed without contraction. Negative constructs (with #) will be referred to as *restrictions* or *negations* [Turchin 74,86].

The *SELECT* construct {...} has the usual semantics: the first applicable path is selected during its evaluation. The applicability of a path is a successful completion of all constituent constructs in it. An unsuccessful evaluation of any clash in a path results in *fail* situation on this path. The next path is evaluated in this case. If an applicable path is found - the construct is successfully completed, otherwise its final result is failure.

An *ALL* construct [...] is new for Refal, its semantics can be understood in terms of set-valued computation. The failure of any of evaluation paths does not influence others. The result of such an execution is a set of results of all successfully computed paths.

As an example consider reverting function:

```
<Reverse eArg> = eRes
    {   eArg : [], [] : eRes;
        eArg : s1 e2, <Reverse e2> s1 : eRes;
        eArg : (e1)e2,
        <Reverse e2>(<Reverse e1>) : eRes
    }
```

The first path describes the case of the argument being empty, the result is empty too. The second path is carrying the next symbol recursively out of functional brackets, and the last one prescribes what to do if parentheses have been met.

We shall often use *semi-predicates* rather than predicates in our work. A semi-predicate is a predicate yielding some result - usually empty - instead of the TRUE value and running infinitely or failing instead of the FALSE value. [Turchin 87]. They are particularly convenient when the program is intended to be treated by the Supercompiler with its powerful technique of cutting off unreachable branches in an evaluation tree.

Some syntactical sugar: in a semi-predicate definition an equality sign and the output pattern can be omitted, clashes and negations with such semi-predicates can be written without right sides also. Functions defined with a single path can be written 'inline', without complex constructs. Clashes of the form: $e_x : e_1 \ldots e_2$ and repeating e or t variables will be used in the *ALL* constructs. They are interpreted as short-hand notations for the following functions calls correspondingly:

```
<Open-E e_x> = <Open-E-aux ()e_x>
<Open-E-aux (e_a)e_b> = (e_1)(e_2)
     [ e_a : e_1, e_b : e_2;
       e_b : t_u e_w, <Open-E-aux (e_a t_u) e_w> : (e_1)(e_2) ]

<Double-E (e_x)(e_y)>
     { e_x : [], e_y : [];
       e_x : s_1 e_a, e_y : s_1 e_b, <Double-E (e_a)(e_b)>;
       e_x : (e_1)e_a, e_y : (e_2)e_b, <Double-E (e_1)(e_2)>,
                               <Double-E (e_a)(e_b)>
     }
```

Double-T is defined similarly.

The next example is the semi-predicate *Symm* checking the symmetry of an expression:

```
<Is-Symm e_a>
     { e_a : [];
       e_a : s_x;
       e_a : s_x e_b s_x, <Is-Symm e_b>;
       e_a : (e_b), <Is-Symm e_b>;
       e_a : )e_b(), <Is-Symm e_b>;
       e_a : (t_x e_1)e_b(e_2 t_y), <Is-Symm t_x (e_1)e_b(e_2) t_y>
     };
```

Now consider a well-known definition of a formal grammar:

```
E = E + T
E = T
T = T * F
T = F
F =(E)
F = a
```

GenGram is a Refal-R representation of this formal grammar: from the nonterminal E it produces all possible strings corresponding to it. Initial function call for the above grammar is *<GenGram E>*, the function definition is:

```
<GenGram s_g> = e_1
  { s_g : E, [       <GenGram E> '+' <GenGram T> : e_1;
                     <GenGram T> : e_1 ];
    s_g : T, [ <GenGram T> '*' <GenGram F> : e_1;
                     <GenGram F> : e_1 ];
    s_g : F, [ (<GenGram E>) : e_1;
                     'a' : e_1 ];
  };
```

Note, that assignments in the GenGram definition have the form $e_x \ldots e_y : e_1$. In the next section it will be shown, how this generative grammar can be transformed to the parser.

3. INVERTER

Refal was extended to simplify the representation of inverse functions. An easy syntactical procedure for inversion is valid for full inversion. An inversion with respect to some subset of function parameters requires more cumbersome methods.

Beforehand with adding proper contractions or negations all paths in every *SELECT* construct are made independent. The *SELECT* construct may be converted into *ALL* construct after that. This is needed because of the following. Let two paths, p_1 and p_2 have E_i as common possible input, E_{o1} and E_{o2} - as outputs of computation $p_1(E_i)$ and $p_2(E_i)$ respectively. Then the computation of $p_2^{-1}(E_{o2})$ yields E_i, i.e. wrong answer for the function as a whole (if E_{o1} # E_{o2}).

3.1. Full Inversion

The general inversion algorithm can be summarized as follows. Let **Inv** be an inversion operator such that **Inv**[[F]] = F^{-1}. It can be written:

```
Inv[[fd_1;...;fd_n]] = Inv[[fd_1]]; ...; Inv[[fd_n]]
Inv[[<fn exp_1> = exp_2 cc]] =
              <Inv[[fn]] exp_2 > = exp_1 Inv[[cc]]
Inv[[ [p_1;...;p_n] ]] = [ Inv[[p_1]];...;Inv[[p_n]] ]
Inv[[c_1,...,c_n]] = Inv[[c_n]],...,Inv[[c_1]]
Inv[[ #[...] ]] = #[Inv[[...]] ]
Inv[[ <fn exp_1> : exp_2]] = <Inv[[fn]] exp_2> : exp_1
Inv[[exp_1 : exp_2]] = exp_2 : exp_1
                              { exp_1 is not equal to <fn...> }
```

A result of the above algorithm is an approximation of an inverse function. The statement $x \in$ **Inv**[[F]]($F(x)$) holds for it.

The 'function' *Gen-Symm* is inverse with respect to above semi-predicate *Is-Symm*, the generator of symmetric expressions. Its result contains variables as well.

```
<Gen-Symm> = e_symm
  [ [] : e_symm;
    s_x : e_symm;
    <Gen-Symm> : e_b, s_x e_b s_x : e_symm;
    <Gen-Symm> : e_b, (e_b) : e_symm;
    <Gen-Symm> : e_b, () e_b() : e_symm;
    <Gen-Symm> : t_x (e_1) e_b (e_2) t_y,
          (t_x e_1) e_b (e_2 t_y) :: e_symm
  ];
```

How does this program work? At first glance every path starting with *<Gen-Symm>* goes into an infinite cycle. But our semantics is of an unfolding type. Square brackets force the paths to be executed in parallel and the function calls are evaluated by unfolding and substituting the corresponding function bodies.

Now let us try our algorithm in the case of Ackermann's function that is not primitive recursive. An unary number system is used here, N is represented as 01...1 with N of 1.

```
<Ack ($e_x$)($e_y$)> = $e_z$
    {     $e_x$ : '0', $e_y$ '1' : $e_z$ ;
          $e_x$ : $e_{x1}$ '1', $e_y$ : 0,
                 <Ack ($e_{x1}$)('01')> : $e_z$;
          $e_x$ : $e_{x1}$ '1', $e_y$ : $e_{yw}$ '1',
                 <Ack ($e_{x1}$ '1')($e_{yw}$)> : $e_{y1}$,
                    <Ack ($e_{x1}$)($e_{y1}$)> : $e_z$;
    }
```

The *Ack* function has two arguments but at this moment a full inversion is of interest. The following description is produced with the use of the above inverter:

```
<Ack$^{-1}$ $e_z$> = ($e_x$)($e_y$)
   [ $e_z$ : $e_y$ '1', 0 : $e_x$;
     <Ack$^{-1}$ $e_z$> : ($e_{x1}$)('01'),
                  0 : $e_y$, $e_{x1}$ '1' : $e_x$;
     <Ack$^{-1}$ $e_z$> : ($e_{x1}$)($e_{y1}$),
            <Ack$^{-1}$ $e_{y1}$> : ($e_{x1}$ '1')($e_{yw}$),
                  $e_{yw}$ '1' : $e_y$, $e_{x1}$ '1' : $e_x$
   ]
```

Applying the interpreter to the above program for any e_z gives at once the first trivial solution: e_x = '0', e_x '1' = e_z but other solutions for e_z greater than '0111' were awaited for a long time and the job was terminated by 'NOT ENOUGH STORAGE' condition - the result of the exhaustive search, the explosion mentioned above. This program is non-deterministic in its nature - there are several possible results for a single argument.

Parser is the promised inverse function of *GenGram* from the previous section:

```
<Parser $e_I$> = $s_g$
    [ $e_I$ : $e_E$ '+' $e_T$, <Parser $e_E$> : $E$,
                  <Parser $e_T$> : $T$, $E$ : $s_g$;
      <Parser $e_I$> : $T$, $E$ : $s_g$;
      $e_I$ : $e_T$ '*' $e_F$, <Parser $e_T$> : $T$,
                  <Parser $e_T$> : $F$, $T$ : $s_g$;
      <Parser $e_I$> : $F$, $T$ : $s_g$;
      $e_I$ : ($e_E$), <Parser $e_E$> : $E$, $F$ : $s_g$;
      <Parser $e_I$> : 'a', $F$ : $s_g$;
    ]
```

This definition is non-deterministic as well. The clash e_I : e_E '+' e_T, for instance, produces a set of all possible partitioning of the input string e_I, where e_E is of type E, e_T is of type T (it is checked further in the path) and the sign '+' between them. But it is a job for the supercompiler to make it more if not completely deterministic. Such techniques will be considered in the next sections.

3.2. Partial Inversion

The word 'partial' here must not be confused with the same word in 'partial evaluation'. *Partial inversion* is a predicate or function inversion with respect to a subset of their parameters and not a partially completed inversion. To obtain a partially inverted function it is necessary to resolve paths with respect to unknown variables instead of using the above inversion procedure.

Mode analysis (SD-annotation) - is used in functional programming as the first stage in partial evaluation process [Jones 88]. In the context of logic programming a similar method is the main tool of functionality and determinacy detecting in program [Reddy 84; Debray 89]. Two possibilities were suggested to add control to the logic program: *cuts* or other primitives are to be inserted into the logic program [Debray 89; Debray,Warren 89], and translation into another [Reddy 84] or the same [Reddy 86] functional language, properly augmented.

When the transformed program is the goal by itself, the final stage of work before being executed, the first approach may be sufficient. But we consider the transformed text as an intermediate result, as an object for further metacomputation. So we are following the last approach. Mode analysis helps us to construct the proper definition of a partially inverse function from the definition of a direct one. This method was used in [Romanenko 88].

These are definitions of Ack_1^{-1} and Ack_2^{-1}, the functions inverse to *Ack* by the first and second arguments respectively - they can be received formally from the *Ack* definition:

```
<Ack$_1^{-1}$ ($e_z$)($e_y$)> = $e_x$
    [ $e_z$ : $e_y$ 1, 0 : $e_y$;
      $e_y$ : 0, <Ack$_1^{-1}$ ($e_z$)01> : $e_{x1}$, 0 : $e_{x1}$ : $e_x$;
      $e_y$ : $e_{yw}$ 1, <Ack$^{-1}$ $e_z$> : ($e_{x1}$ 1)($e_{y1}$),
            <Ack$_1^{-1}$ ($e_{y1}$)($e_{yw}$ 1)> : $e_{x1}$ 1, $e_{x1}$ 1 : $e_x$
    ];

<Ack$_2^{-1}$ ($e_x$)($e_z$)> = $e_y$
    [ $e_x$ : 0, $e_z$ : $e_y$ 1 ;
      $e_x$ : $e_{x1}$ 1, <Ack$_2^{-1}$ ($e_1$)($e_z$)> : 01, 0 : $e_y$ ;
      $e_x$ : $e_{x1}$ 1, <Ack$_2^{-1}$ ($e_{x1}$)($e_z$)> : $e_{y1}$ ,
            <Ack$_2^{-1}$ ($e_{x1}$ 1)($e_{y1}$)> : $e_{yw}$, $e_{yw}$ 1 : $e_y$
    ];
```

Because of initial independence of paths extra negations for them in the function *Ack* are not needed. The paths independence for the function $Ack2^{-1}$ can be proved easy, the way for it will be shown below. So in this case the *ALL* construct can be replaced by the *SELECT* one. But the function $Ack1^{-1}$ is not so simple: it uses Ack^{-1} in its definition with all indeterminacy features of the last.

4. METACOMPUTATION

4.1. Supercompiler

A tree of generalized histories of an evaluation is constructed during driving. This tree is called *the graph of states*, its nodes are general expressions, they are called *configurations*. The edges of the graph are segments - they contain matching operations: contractions, restrictions, assignments. The process of driving may be infinite, hence, the looping back of all potentially infinite branches is needed to receive a self-sufficient finite residual program. The often necessary condition to loop back the current configurations is generalization.

In the previous version of the supercompiler configurations contained free variables taking arbitrary values subject to specified restrictions. These restrictions had a form of *contractions* or elementary *negations*. In the last version Turchin added *set filters*. The constraints imposed on free variables can be described now as total recursive semi-predicates, i.e. semi-predicates that always terminate. They are function definitions, its result is not relevant, their successful completion is considered as TRUE and the failure as FALSE. The main supercompiler principle of top-down approximation is right for filters as well. If the filter says 'YES' then it may be, but if it says 'NO' then it is necessary 'no'! Note the special meaning of the term 'total' in this context [Turchin 90].

Consider nested function calls: <*F* <*G* exp > >. The supercompiler can treat such a configuration as a whole and may even reduce two or multi-pass algorithms to one-pass. But sometimes decomposition is required: <*G* exp > : e_{New}, <*F* e_{New}> .... The information about the structure of e_{New} is not transferred to the next call. For the time being you may *manually* insert a filter describing the features of e_{New}, for instance, that it does not contain letters "A". A task of *automatic filter generation* emerges as a result, and the program inversion helps to solve this problem.

Let *G* above replaces all instances of 'A' by 'B' and *F* replaces all instances of 'A' by 'C'. The nested function call fails in any case. But after decomposition the residual program will contain both definitions of *F* and *G*. Filter generation produces from *G*, inverting it, the proper constraints for e_{New} such, that their combination with the *F* call will result in falure and eliminating this path.

4.2. Inverting the history of computation

Examination and transformation of computation history can be useful in constructing more efficient inverse functions than the general inversion procedure may produce. Turchin used the 'Orwellian principle of permanently rewriting the history' in the supercompiler for generalization purposes [Turchin 88]. We shall use the principle of turning the history backwards for inversion purposes. This techniques may be a constituent part of the supercompiler itself.

Consider a computation sequence for the function *Ack*:

```
<Ack ($e_{x0}$)($e_{y0}$)>
    . . .
<Ack ('0')($e_{yz}$)>
    $e_{yz}$ '1'
```

Every intermediate configuration in this sequence has a form of: <*Ack* . . . >, where argument may contain other calls to *Ack*. A function *Step* can be constructed, describing the argument transformation in a step of this sequence. Only recursive branches of *Ack* definition are treated and the outer calls to *Ack* in them are eliminated:

```
<Step ($e_x$)($e_y$)> = ($e_{x1}$)($e_{y1}$)
    { $e_x$ : $e_{x1}$ '1', $e_y$ : 0, '01' : $e_{y1}$;
      $e_x$ : $e_{x1}$ '1', $e_y$ : $e_{yw}$ '1',
          <Ack ($e_{x1}$ '1')($e_{yw}$)> : $e_{y1}$
    }
```

This function defines a computation step for *Ack*. One can describe the inverse function for *Ack* turning the above sequence backwards. First, the contraction e_z : e_{yz} '1' evaluates, an initial value of e_x becomes '0'; then the following sequence of configurations will constitute the inverse computation history:

```
<Ack-x ('0')($e_{yz}$)>
    . . .
<Ack-x ($e_{x0}$)($e_{y0}$)>
```

A new name *Ack-x* is used for an intermediate function, the resulting Ack^{-1} will contain the initial format transformation and a call to *Ack-x*. The inversion of the *Step* function is obtained easily with the use of the inverter:

```
<$Step^{-1}$ ($e_{x1}$)($e_{y1}$)> = ($e_x$)($e_y$)
    [ $e_{y1}$ : '01', '0' : $e_y$, $e_{x1}$ '1' : $e_x$;
      <$Ack2^{-1}$ ($e_{x1}$ '1')($e_{y1}$)> : $e_{yw}$,
          $e_{yw}$ '1' : $e_y$, $e_{x1}$ '1' : $e_x$
    ]
```

It turned out, that *Step* is 'better' for inverting, than *Ack* - it uses deterministic $Ack2^{-1}$ instead of non-deterministic Ack^{-1}, and moreover, one can show that its paths are independent, so $Step^{-1}$ is a deterministic function. We can put a new envelope 'Ack^{-1}' on the function $Step^{-1}$ and obtain the final self-sufficient definition of Ack^{-1}:

```
<Ack⁻¹ ez> = (ex)(ey)
      { ez : eyz '1', <Ack-x ('0')(eyz)> : (ex)(ey) };
<Ack-x (ex1)(ey1)> = (ex)(ey)
      [ (ex1)(ey1) : (ex)(ey) ;
        ey1 : '01', '0' : ey2, ex1 '1' : ex2,
                  <Ack-x (ex2)(ey2))> : (ex)(ey);
       <Ack2⁻¹ (ex1 '1')(ey1)> : eyw,
                  eyw '1' : ey2, ex1 '1' : ex2,
                  <Ack-x (ex2)(ey2))> : (ex)(ey)
      ]
```

This definition is quite different from its previous namesake, in spite of that they are both non-deterministic. It produces rather quickly, when executed, all possible correct pairs of $(e_x)(e_y)$ before its failure.

Partially inverted variations of *Ack* can be pulled out of this description. They have a form:

```
<Ack1⁻¹ (ez)(ey)> = ex
      { ez : eyz '1', <Ack-x1 ('0')(eyz)(ey)> : ex };
<Ack-x1 (ex1)(ey1)(ey)> = ex
      [ (ex1)(ey1) : (ex)(ey) ;
        ey1 : '01', '0' : ey2, ex1 '1' : ex2,
                  <Ack-x1 (ex2)(ey2))(ey)> : ex;
       <Ack2⁻¹ (ex1 '1')(ey1)> : eyw,
                  eyw '1' : ey2, ex1 '1' : ex2,
                  <Ack-x1 (ex2)(ey2))(ey)> : ex
      ]

<Ack2⁻¹ (ex)(ez)> = ey
      { ez : eyz '1', <Ack-x2 ('0')(eyz)(ex)> : ey };
<Ack-x2 (ex1)(ey1)(ex)> = ey
      [ (ex1)(ey1) : (ex)(ey) ;
        ey1 : '01', '0' : ey2, ex1 '1' : ex2,
                  <Ack-x2 (ex2)(ey2))(ex)> : ey;
       <Ack2⁻¹ (ex1 '1')(ey1)> : eyw,
                  eyw '1' : ey2, ex1 '1' : ex2,
                  <Ack-x2 (ex2)(ey2))(ex)> : ey
      ]
```

These new definitions are not equivalent to its namesake in the previous section and are much more efficient. For example, the computation of the Ack_2^{-1} from the previous section with arguments: $e_x = 3$ and $e_z = 253$ lasted 34 min 21 sec (337427 steps of Refal) and the latter has done this work for 3 min 53 sec (42443 steps of Refal) on some IBM PC type computer. Note that the clash $(e_{x1})(e_{y1}) : (e_x)(e_y)$ in the definition of *Ack-x2* is not a simple assignment of the known $(e_{x1})(e_{y1})$ to an unknown $(e_x)(e_y)$. The equivalence of both known e_{x1} and e_x are checked here in contrast to the *Ack-x* definition. This is the source of a Ack_2^{-1} determinacy.

The relative simplicity of the *Step* function comparing to the original one is the main criterion of applicability of the above method. Any function can be transformed into the proper form with the help of an extra parameter, representing the accumulating result of the function given. Consider this transformation using a simple primitive recursion scheme as an example:

$$h(0) = C$$
$$h(y+1) = g(y,h(y))$$

Let z be a such parameter. Its initial value is undefined, we shall use the symbol $\perp$ to underline this fact. The goal is to construct such a new function description that contains a model of the given function.

$$h(y) = H(y,\perp)$$
$$H(0,\perp) = C$$
$$H(y+1,\perp) = H(y,H(y,\perp))$$
$$H(y+1,z) = g(y,z)$$

z does not contain $\perp$, hence the last two sentences are independent. The essence of the transformation is replacing all internal recursive calls in the right parts of the definition by the call to the new function, instantiated with $\perp$ as a value for the extra parameter. For every recursive sentence its analogue appears with the new variable instead of recursive calls. Now, omitting the corresponding results, similar to those in the previous case, our last inverting method may be used to function H.

This transformation generally adds an extra level of recursion to a function definition, the method of history inverting takes off one recursion level. So we were lucky in the case of *Ack*, where such a transformation was not needed.

4.3. Determinacy detection

The supercompiler can optimize the resulting programs of either of inversion methods. It was said above that a function approximation is obtained as a result of general inversion. Determinacy detection and *ALL-to-SELECT transformation* can be obtained as a result of inverted program driving. Generally, an *ALL* construct is less efficient than a *SELECT* construct with the same body. Hence, it may be very important for program optimization, if conditions can be found for an *ALL* construct being transformed into a *SELECT* construct. In general case functionality of *ALL* construct is unresolvable but sometimes it may be tested in a finite number of steps by driving.

A special type of driving was developed for *ALL* constructs: the domain of some path is being used as an argument in the initial driving configurations for other paths. Driving is executed over all lower paths of the construct. It can be shown with the $Ack2^{-1}$ function as an example. Its second sentence contains contraction e_{y1} : '01'. It can be ascribed from the left to the third sentence, then the function call <$Ack2^{-1}$ (e_{x1} '1')('01')> is to be executed. Only the third sentence of the function $Ack2^{-1}$ is applicable at this time. The next call to be evaluated is <$Ack2^{-1}$ (e_{x1} '1')('0')>. At the first step it fails.

4.4. Inverse Driving

The program can be improved with the use of some additional information about its arguments, but the supercompiler can benefit from known partially or completely its results. We shall refer to such procedure as *inverse driving* (this term was suggested by V.Turchin). The inverse driving or specialization by partially known result is particularly efficient in the case of generators such as *Gen-Symm* defined in the second section. In our context this method can be described as follows.

Let a clash <*F* ...> : exp appears where exp is more narrow than an output pattern F_{out} of the function *F*, then a new function F_x can be constructed from the old one if we ascribe a clash F_{out} : exp to every path of the function *F* and run the properly augmented supercompiler. It will do its usual work at that time: cleaning-up the contradictory paths.

Suppose that the first and the last characters of *Gen-Symm* output are both 'A', i.e.:

<*Gen-Symm* > : 'A' e_s 'A'

The obvious transformations of *Gen-Symm* definition with the above restrictions on its result produces the specialized version of this definition - denoted by *Gen-Symm_AA*:

```
<Gen-Symm_AA > = 'A' e_s 'A' ,
    <Gen-Symm > : e_s
```

Of six paths of *Gen-Symm* only a single path remained. Because the main recusive scheme of *Gen-Symm* is still working one step is saved as a result. This example is very simple but one may imagine more complex constrains with more deep transformations by the supercompiler.

Driving *Is-Symm* with the partially known argument 'A'e_s'A' yields immediately:

```
<Is-Symm_AA e_s > ,
   { <Is-Symm e_s > };
```

the mirror of the *Gen-Symm_AA* definition.

5. IMPLEMENTATION

The interpreter for Refal-R and inverter were implemented with the use of programming language Refal Plus. Refal Plus is designed by Sergei Romanenko and Ruten Gurin and was implemented on IBM PC type computers. The supercompiler system is implemented by Valentin Turchin with some collaborators in the programming environment of Refal 5. Integration of the supercompiler and Refal-R described in the paper is under development now.

6. CONCLUSION

Extensions to the functional language Refal were proposed to make convenient a natural expression both direct and inverse semantics of functions. The special versions of the direct and inverse driving is suggested for function optimization. The embedding of the inversion operation into the programming language will result in a new view on the connection between forward and backward analysis of functional programs, they are becoming supplementary for direct and inverse ones. The forward analysis of a direct function strongly corresponds to the backward analysis of the inverse one.

The most important task now is to combine the current version of the supercompiler and the Refal-R implementation with inversion methods. Such a combination may become a very powerful programming system integrating the functional and logical approaches in programming.

It is naturally to ask about the supercompiler or a partial evaluator: 'What can it do now?' - the answer usually is: 'Oh, it can manage nested function calls now!' or 'It can threat higher order functions already!'. At first sight the function inversion is not a method of equivalent function transformation. But the supercompiler builds a model of a program and tries to optimize it during metacomputation by all possible means. Arguments and/or results of some function may be known to some extent and the supercompiler can perform the partial or full inversion before the driving continues. Such approach resembles relative program specification [Runciman,Jagger 90]. So, the inversion procedures make a substantial enrichment of the supercompiler and manipulation of the history in computer science, but not in real life, is a very useful thing.

ACKNOWLEDGMENTS

Several people have contributed to this work. I want to thank the members of Refal-group in Moscow for being my opponents in discussion about the various extensions of the language. Sergey Abramov's URA implementation was the initial inspiration of my work. But especially I thank Andy Klimov, the talks with whom and whose brilliant questions were the main source for many ideas of this work. I had fruitful discussions with Neil Jones who made some invaluable helpful remarks about my previous vague publication.

Great thanks to referee of this symposium, who made a lot of work trying to improve my language and contents.

This work could not be carried out without the permanent collaboration with my teacher Valentin Turchin in spite of the geographical distance. His works on the inverse problem, supercompilation and cybernetic foundation of mathematics inspired all my movement in this area.

REFERENCES

[Abramov 88]
S.M.Abramov. Metacomputation and Logic Programming. In *Semiotic aspects of formalization of the intellectual activity*, All-union school-workshop "Borjomi-88", Moscow, 1988.(In Russian).

[Abramov 89]
S.M.Abramov. Metacalculations and Logical Programming, 1989, Unpublished.

[Cutland 80]
N.Cutland. COMPUTABILITY. An Introduction to recursive function theory, Cambridge University Press, 1980.

[Debray 89]
S.K.Debray. Static Inference of Modes and Data Dependencies in Logic Programs. In *ACM TOPLAS*, Vol.11,No.3, July 1989,418-450.

[Debray, Warren 89]
S.K.Debray, D.S.Warren. Functional Computations in Logic Programs. In *ACM TOPLAS*, Vol.11,No.3, July 1989,451-481.

[Fuller,Abramsky 88]
D.A.Fuller, S.Abramsky. Mixed Computation of Prolog Program. In *New Generation Computing*, 6(1988) 119-141.

[Futamura 71]
Y.Futamura. Partial Evaluation of Computation Process - an Approach to a Compiler-Compiler. In *Systerms, Computers, Control*. Vol. 2, No. 5, 1971, pp.45-50.

[Glueck Turchin 90]
R.Glueck, V.Turchin. Application of Metasystem Transition to Function Inversion and Transformation. In *Proceedings of International Symposium on Symbolic and Algebraic Computation* (ISSAC'90), 1990.

[Gris 81]
D.Gris. The Science of Programming. Springer-Verlag, 1981.

[Harrison 88]
P.G.Harrison. Function Inversion. In [Jones,Bjorner,Ershov 88], pp.153-166.

[Hughes,O'Donnell 89]
J.Hughes, J. O'Donnell. Expression and Reasoning About Non-deterministic Functional Programs. In *Functional Programming*. K.Davis, J.Hughes - eds. Glasgo, 1989, pp. 308-328.

[Jones,Bjorner,Ershov 88]
Proceedings of the Workshop on Partial Evaluation and Mixed Computation. N.Jones, D.Bjorner, A.Ershov - eds. Gammel Avernaes, Denmark, 18-24 October 1987, North-Holland,1988.

[Jones 88]
Automatic Program Specialization: a Reexamination from Basic Principles. In [Jones,Bjorner,Ershov 88], pp.225-282.

[Kursawe 88]
Pure Partial Evaluation and Instantiation. In [Jones,Bjorner,Ershov 88], pp.283-298.

[McCarthy 56]
J.McCarthy. The Inversion of Functions defined by Turing Machines. In *Automata Studies*. C.E.Shannon, J.McCarthy - eds. Princeton, 1956. pp.177-181.

[Reddy 84]
U.S.Reddy. Transformation of Logic Programs into Functional Programs. In *Intern. Symp. Logic Prog.*, IEEE, 1984, pp.187-197.

[Reddy 86]
U.S.Reddy. Logic Languages based on Functions: Semantics and Implementation. Technical Report UIUC-DCS-R-86-1305, Illinois, 1986, Ph.D.thesis done at Univ. of Utah.

[Reddy 87]
U.S.Reddy. Functional logic languages, Part 1. - In *Graph Reduction*, Springer-Verlag, 1987, pp.401-425. (LNCS, Vol. 279)

[Romanenko A. 88]
A.Y.Romanenko. The Generation of Inverse Functions in Refal. In [Jones,Bjorner,Ershov 88], pp 427-444.

[Runciman,Jagger 90]
C.Runciman,N.Jagger. Relative Specification and Transformational Re-use of Functional Programs. In *Lisp and Symbolic Computation: An international Journal*, 3,1990, pp.21-37.

[Shoman, McDermott 84]
Y.Shoman,D.V.McDermott. Directed Relations and Inversion of Prolog Programs. - in *Proceedings of the International Conference of Fifth Generation Computer Systems*. ICOT,1984.

[Slagle 74]
J.Slagle. Automated Theorem-Proving for theories with simplifiers, commutativity and assciativity. In *JACM*, Vol. 21, No. 4, Oct.1974, pp.622-642.

[Turchin 72]
V.F.Turchin. Equivalent Transformations of Recursive Functions defined in REFAL. - in *Teoria yazykov i metody postroenia system programirowania*. Trudy sympos. Kiev-Alushta,1972,pp.31-42.(In Russian)

[Turchin 74]
V.F.Turchin. Equivalent Transformations of Programs in REFAL. CNIPIASS, Gosstroy SSSR, Moscow, 1974 (in Russian)

[Turchin 79]
Turchin,V.F.,Supercompiler System Based on the Language REFAL. - SIGPLAN Notices, 14(2), 1979, pp.46-54.

[Turchin 80]
V.F.Turchin. The Language Refal. The Theory of Compilation and Metasystem Analysis. Curant Institute of Mathematics. Technical report #018, NY, 1980.

[Turchin 86]
V.F.Turchin. The Concept of a Supercompiler. In *ACM TOPLAS*, Vol.8, No. 3, 1986, pp 292-325.

[Turchin 87]
V.F.Turchin. A Constructive Interpretation of the Full Set Theory. In *The Journal of Symbolic Logic*, Vol. 52, No 1, March 1987, pp.172-201.

[Turchin 88]
V.F.Turchin. The Algorithm of Generalization in the Supercompiler. In [Jones,Bjorner,Ershov 88], pp 531-549.

[Turchin 89]
V.F.Turchin. REFAL-5. Programming Guide and Reference Manual. Refal Systems Inc., 1989.

[Turchin 90]
V.F.Turchin. The supercompiler system. 1990. Unpublished.