Compiling Flang

Andrei Mantsivoda, Vyacheslav Petukhin
Department of Mathematics, Irkutsk University
Irkutsk 664003, Russia
email: kvant!mant@glas.apc.org

Abstract

The problems of compilation of a functional-logic language Flang are considered. A new phase of compilation - global dataflow analysis - is discussed. It is shown that this phase can extremely improve the performance of a produced code. For some benchmarks a code which is produced by the Flang compiler has about the same speed as the speed of corresponding Pascal programs.

1 Introduction

In this paper we consider the main aspects of implementation of a functional-logic language Flang [1]. We use *the Warren Abstract Machine* (WAM) [2] as the basic technique for this purpose, because WAM is most convenient for implementation of Flang's main tools. Unfortunately, the 'pure' WAM does not have all means necessary for implementation of the language, since it does not support such important features of Flang as, say, functionality. In order to adapt WAM to our purpose we change WAM itself and introduce some transformations of Flang source programs to make them 'edible' by WAM. This modification of WAM is called the *Flang Abstract Machine* (FAM). The main additional features of FAM are

- a support of valued terms;
- optimizations for parameters management of a functional style;
- implementation of a special technique for algebraic computations;
- a support of a constraint satisfaction technique [3];
- additional instructions to support special tools (*e.g.* conditional expressions);

The Flang compilation process consists of four principal stages. The first phase is translation of a Flang source program into some simplified form. After this, the compiler fulfills very significant and quite new step — the global dataflow analysis of a program.

This step allows to increase the performance of a produced code drastically. The next step is the translation of a Flang program into the intermediate code of FAM. In the last phase the compiler translates this intermediate code to a native code of a target computer.

The problem of the efficient implementation of functional and logic languages draws the attention of many computer scientists [2,4-8]. Our general approach of compilation is close to approaches of Aquarius Prolog [5] and Parma [6]. The main diferences are

- Flang is an extension to Prolog containing additional features and tools which should be translated by a Flang compiler;
- Aquarius and Parma translate Prolog programs into intermediate languages which close to a code of a real computer; on the other hand, we prefer to work with the modification of WAM;
- As a target computer we use IBM PC which has the relatively weak processor. On the other hand, those optimizations that useful for large computers are not always the best for smaller machines.

(the last item does not mean that FAM can not be efficiently implemented on large computers).

The compiler translating a Flang program into the native code of the IBM PC has been implemented in C. The performance of an executable code is very high. For example, for some benchmarks the performance of Flang programs is close to corresponding programs written in TurboPascal. On some benchmarks the Flang compiler generates a code which 8-10 times faster than a code generated by the Arity/Prolog compiler (the *N-Queens* problem, some sorting algorithms, *etc.*).

2 Description of Flang

Flang [1] is a functional-logic language intended for symbolic data processing, solving combinatorial problems, for algebraic computations, as well as for training students in methods of artificial intelligence. Flang makes it easy to write programs in functional, logic and integrated styles. The reason is that these two kinds of programming are unified in Flang on some natural basis.

Flang can be considered as an extension to Prolog, but really it came from the functional style of programming. It is impossible to separate logic and functional parts of Flang: there is one idea for the two styles.

Flang contains two main types of functions:

- non-deterministic functions;
- algebraic functions;

There is the following distinction between them. When the Flang machine fails to reduce a non-deterministic goal it uses the backtracking procedure, but when a goal is algebraic, the system uses a kind of self-quotation (leaves the goal unreduced).

Non-deterministic functions are the generalization of 'usual' functions in the following way:

- the evalution of functions with unground arguments is permitted;
- the depth-first strategy of computation of functions is used: if the system can not reduce a goal, it uses the backtracking procedure looking for alternative ways of execution;

This generalization of functions includes usual Prolog relations (they are represented by functions with the only one value — *true*). On the other hand, we can treat non-deterministic functions as usual functions and, thus, to write purely functional programs. Let us consider some examples of definitions in Flang. We begin with purely functional definitions. The first of them is *factorial*:
$factorial(0) <=> 1;$
$factorial(X) <= X > 0 \;\; X * factorial(X-1);$

The next one is *append*:
$append([], X) <=> X;$
$append([X|Y], Z) <= [X|append(Y, Z)];$

The naïve *reverse* can be defined as follows:
$reverse([]) <=> [];$
$reverse([X|Y]) <= append(reverse(Y), [X]);$

Logic definitions:
$parent(paul, john) <= true;$
$parent(john, george) <= true;$
$grandparent(X, Y) <= parent(X, Z) \;\; parent(Z, Y);$
This program is equivalent to the following program in Prolog:
$parent(paul, john).$
$parent(john, george).$
$grandparent(X, Y) : -parent(X, Z), \;\; parent(Z, Y).$

The *QuickSort* program is an example of an integrated style definition:
$partition(X, [], [], []) <=> true;$
$partition(X, [Y|Z], [Y|W1], W2) <= X >= Y- > partition(X, Z, W1, W2);$
$partition(X, [Y|Z], W1, [Y|W2]) <= partition(X, Z, W1, W2);$
$qsort([], X) <=> X;$
$qsort([X|Y], Z) <= partition(X, Y, W1, W2) \;\; qsort(W1, [X|qsort(W2, Z)]);$
The relation *partition* is defined in the logic style. The definition of *qsort* is functional. But the right part of its second rule contains logic variables $W1$ and $W2$ which are absent in the head of the rule.

The next program is the example of a functional-logic definition:
$ancient(X, Y) <= parent(X, Y) \;\; [X, Y];$

ancient(X,Y) <= *parent*(X,Z) $[X|$*ancient*$(Z,Y)]$;
The function *ancient* returns the list of relatives which are between the ancient X and the offspring Y in the genealogical tree. While computing this function and searching through the genealogical tree, the Flang system can use backtracking – the action which is quite unusual in the functional programming.

Some remarks on the programs: the atom '<=' plays in Flang the role of Prolog's ':-'; '– >' is the symbol for *cut* in Flang; '<=>' is a shorthand for '<= – >' *(which is equivalent to* ': –!' in Prolog). The function *factorial* has the purely functional definition. The definition of *qsort* is integrated – it does not need backtracking, but the variables $W1$ and $W2$ in the rules are logic and the system uses unification to treat them. The relations *parent* and *grandparent* are defined just like those in Prolog. The function *ancient* is non-deterministic (the system can use the backtracking procedure while executing it).

Algebraic functions are introduced in Flang to facilitate writing programs in the field of algebraic computations. The distinction between usual (non-deterministic) and algebraic functions can be demonstrated in the following simple example. Let us define a function f:
$f(0) <= 1$;

This function is defined only on 0. The distinction between the situations when f is usual and when it is declared as the algebraic function is shown in the following table :

f is non-deterministic		*f is algebraic*	
Goal	*f(0)*	*Goal*	*f(0)*
Answer	*1*	*Answer*	*1*
Goal	*f(1)*	*Goal*	*f(1)*
Answer	fail	*Answer*	f(1)

The following example demonstrates a dialogue between a user and the Flang system when '+' and '*' are declared as algebraic functions:

Goal	$(a+1)*b$;
Answer	$(a+1)*b$
Rule	$(X+Y)*Z <= X*Z+Y*Z$;
Goal	$(a+1)*b$;
Answer	$a*b+1*b$
Rule	$1*X <= X$;
Goal	$(a+1)*b$;
Answer	$a*b+b$
and so on	

An algebraic function has properties of a function and a structure at the same time.

Sometimes it behaves as a function and sometimes as a constructor of a structure. Using this feature of an algebraic function we can define in Flang constructors with constraints, for instance, the constructor of *an ordered list.* Let us declare the atom '.' as an algebraic function and as an infix right associative operator. The sorting program consists of only one logically pure and simple rule:
$X.Y.Z <= X > Y -> Y.(X.Z);$
Now,

Goal	3.2.1.6.5.4.9.8.7.0.*nil*;
Answer	0.1.2.3.4.5.6.7.8.9.*nil*;

The idea of an algebraic function plays in Flang about the same role as the idea of *an unfree algebra* (an algebraic data type with associated laws) in D.Turner's *Miranda* [7]. But the concept of an algebraic function seems to be closer to the kernel of Flang than unfree algebras to the kernel of Miranda.

Algebraic functions can be compiled into extremely fast code. For example, the sorting program above is compiled into the native code of IBM PC which runs with about the same speed as the list sorting program written in TurboPascal and having the same procedural semantics. The only difference between two programs is that the Flang system wastes time to dynamically allocate elements of the lists in the Heap. On the other hand, in the Pascal program the destructive assignment is used which allows to utilize old lists for building new ones. The further discussion on this problem can be found in the next sections of the paper.

Recently, tools for solving combinatorial problems were incorporated into Flang. We used some modification of the methods offered in [3]. Special instructions and new tools for memory management which support constraint satisfaction technique were introduced in FAM. But it is the matter for other publications.

3 Outline of compilation process

In this section we describe the main steps of compilation of Flang programs. The compiler fulfills the following steps:

- translation of a Flang source program into a standard form;
- global dataflow analysis;
- translation of a transformed Flang program into the intermediate code of FAM;
- translation to the native code of a target computer.

The first step of the compilation process is ***transformation of a Flang source program into the standard form.*** We demonstrate this step, using the definition of the function *factorial*
$fact(0) <=> 1;$
$fact(x) <= x > 0 \;\; x * fact(x-1);$

The main hereditary defect of FAM is that it can not manipulate nested calls of functions. So, before translating into the FAM code, the compiler has to transform the source program to get rid of terms of the form $f(\ldots g(\ldots)\ldots)$, where the call of the function g is the argument of the call of f. In the definition of *factorial* the compiler transforms the term $X * fact(X - 1)$. This procedure of deliverance from nested calls is known as *flattening* [4].

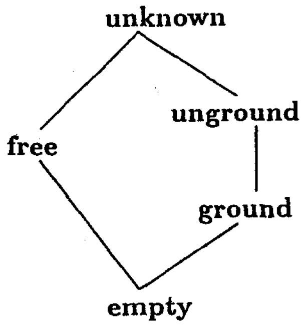
To demonstrate the main idea of flattening we apply the standard Prolog built-in relation *is*. Using it, the second rule of the definition will be transformed into
$fact(x) <= x > 0, \_v1 \ is \ x - 1, \_v2 \ is \ fact(\_v1), \_v3 \ is \ x * \_v2, \_v3;$
where $\_v1, \_v2$ and $\_v3$ are some new variables. The result of computation is saved in the variable $\_v3$. In general case, given a term of the form $f(\ldots g(\ldots)\ldots)$, the compiler transforms it into something like
$\_v \ is \ g(\ldots)\ldots f(\ldots \_v \ldots)$
where $\_v$ is a new variable. Variables like $\_v$ have special features which allow to fulfill some important optimizations.

4 Global dataflow analysis

The global analysis of a Flang program contains the following steps:

- Analysis of arguments.
- Analysis of choice points in a program.
- Analysis and separation of functions and predicates.
- Choosing methods of returning values for 'output' arguments.

The first step (*the analysis of arguments*) gives the information which is very important for optimizations. For all functions of a compiled Flang program, the analyzer computes types of arguments. We can choose many different algorithms for type computations with different levels of complexity (some of them were described in [5, 6]). In the current version of the Flang compiler we use the following simple lattice of types:



In this lattice:

- *free* - an argument is a free variable; it means that all occurences of the argument are free.
- *unground* - the argument is a term containing free variables.
- *ground* - no occurences of free variables in the argument.
- *unknown* - different occurences of the argument have different types and at least one of them has the type free (otherwise, the type of the argument is *unground*).

In our compiler we use the fast but at the same time quite powerful algorithm for this part of the global analysis. This algorithm resembles *the abstract interpretation* and can be named 'tracing free variables'. The information received from this step of the global analysis is very significant. For example, when all variables of a program are either *free* or *ground*, then this program can be executed without the use of the *Trail.*

The next step is the analysis of *choice points of a program.* During the analysis of choice points, for any function of a program the Flang compiler receives the information whether this function deterministic or not. The information that some variables of a program have the type *ground* or *unground* let the system make more refined analysis of choice points and reduce their number.

The system also analyzes whether *the returned value of a function is used anywhere in the program or not.* This analysis permits not to lose time returning unnecessary values when a function plays the role of a predicate.

The last part of the global analysis chooses *methods to return values through 'output' arguments.* In Flang there are two ways to return values. Firstly, we can use free variables (like in Prolog). Secondly, functions themselves have their own values (like in functional programming). In both cases, the compiler applies one of two different methods of returning values during execution of a program. There are *return-by-value* and *return-by-pointer* tecniques (the second one is the standard Prolog method). The first tecnique is more simple and generates less number of references. But it makes impossible the tail recursion optimization. So, in the compiler the following scheme is used. The compiler recognizes all rules were the tail recursion optimization can be used and only for these rules *the return-by-pointer technique* is applied.

The most essential information which is received from the global analysis for each function f of arity n, the definition of which consists of m rules, contains the following data:

$$< (t_0, t_1, t_2, \ldots, t_n), p, d, c, (tr_1, d_1, c_1), \ldots, (tr_m, d_m, c_m) >,$$

where
$t_{i,i=\overline{0,n}} \in \{ground, unground, free - by - value, free - by - pointer, unknown\}$;
$p \in \{function, predicate\}$;
$d, d_1, \ldots, d_m \in \{determ, non - determ\}$;
$c, c_1, \ldots, c_m \in \{set - choice - point, no - choice - point\}$;
$tr_1, \ldots, tr_m \in \{tail - recursion - is - possible, tail - recursion - is - impossible\}$

5 Architecture of FAM

The Flang Abstract Machine (FAM) is an extension and a modification of WAM [2]. It was designed by the authors as the basis of the Flang compiler. In this section we describe FAM following the terminology from [2].

The main data areas of FAM are

- *Stack* (Local Stack)
- *Heap* (Global Stack)
- *Trail*
- The area for *Registers*

A state of FAM depends on the following registers:

P - program pointer

E - last environment

B - last coice point

A - top of stack

Tr - top of trail

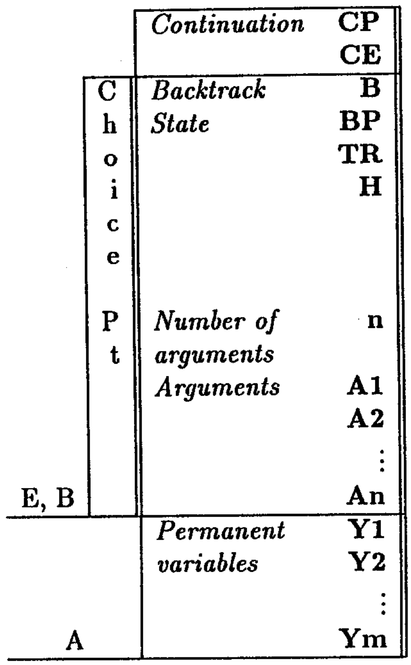
H - top of heap

M - mode of unification (write/read)

S - structure pointer

R1, R2,...,Rn - registers for passing parameters

The permanent variables will be denoted by $Y1, \ldots, Ym$. To simplify instructions we also use a register **BP** (backtracking program pointer). It corresponds to nothing in real execution and can be characterized as the register of the compilation time. The ***activation record*** in the *Stack* of FAM has the following form:



We did not intend to design an abstract machine which is completely independent of the architecture of a real computer. The problem is that the type of an architecture has an effect not only upon the efficiency of FAM instructions. Different kinds of an architecture can leed to different optimization principles. But there is the invariant part of FAM which is the same for all kinds of computer architectures.

We have implemented the version of FAM for IBM PC. Since this computer has very small number of hardware registers and these registers are specialized, the Flang compiler does not allocate the main FAM registers dynamically, but use some hardware registers for them. For instance, **P** is allocated in **IP**, **E** in **BP**, **A** in **SP**, **H** in **BX**. Other registers are allocated in the operative memory.

To improve the perfomance of the produced code in the case of arithmetic computations, the system uses one more register **T** (the temporary accumulator register). It is allocated in the hardware registers **AX** and **CX**. So, the set of FAM instructions is extended by special instructions dealing with this special register.

We see that in the case of IBM PC, such principle of optimization as the minimization of the number of registers **R1**,...,**Rn** is quite inessential. On the other hand, the possibility to group together instructions having occurences of the same variable is very important, since it allows to use the register **T** most efficiently.

6 Compilation into FAM

The compilation into FAM goes independently for each rule of a Flang program (with the use of the information received from the global analysis). To describe the compilation process we should introduce the notions of the left and the right parts of a rule. The left part of a rule is the head of the rule and all goals before the first user-defined goal. For instance:

$$\underbrace{fact(X) <= X > 0, V1 = X - 1,}_{left}\underbrace{V2 = fact(V1), V2 * X;}_{right}$$

The module of the Flang compiler translating a single rule consequently fulfills the following steps:

- analysis of unifications in the head of the rule
- generation of an appropriate *try*-instruction for the rule
- generation of *get*-instructions
- generation of instructions for environment allocation
- analysis of operations and registers manipulations that should be fulfilled between the calls of goals in the right part of the rule and generation of corresponding instructions

The important property of the compiler is capability to avoid choice point creation in the code of a deterministic function. The Flang compiler minimizes the number of choice point creations and uses only necessary part of choice points when it is possible.

The careful work with choice points allows to use three different types of backtracking:

- *branching* - no need to restore values and no choice point
- *near backtracking* - the system restores the Heap and the Trail states but not registers
- *far backtracking* - the standard full backtracking.

To illustrate the process of choosing the appropriate type of backtracking let us introduce the following notion. We say that the unification is *complicated* if it causes the growing of the Heap and/or the Trail. Using the information given by the global analysis, the compiler of a rule counts the number of complicated unifications. Now, let one of the following conditions holds:

- there is no complicated unifications;
- the rule is deterministic and contains only one complicated unification which can be fulfilled after all other unifications.

Then in the code of this rule the *branching* is applied and a choice point should not be created until computation of the left part of the rule is finished.

In the conclusion of this section several examples of FAM-instructions are described. To explain the semantics of the instructions we use the terminology from [2]. In the following, any variable R has the form $Var(t,v)$, where $t \in \{atom, int, str, ref\}$ and $t = Tag(R)$, $v = Value(R)$. $Ref(Rn)$ denotes an object which address is allocated in *Rn*. The compilation time operations are concluded in square brackets *'['* and *']'*. We hope that almost all instructions below are self-evident. The group of instructions *det...* is used to compile deterministic definitions. The instruction *get_atom An Rm* is used when the global analysis shows that the argument *Rn* has the type *ground* (otherwise, the instruction *get_atom_first* is applied). In *get_structure*, *Rn* should have the type *ground* too.

$Deref(Rn)$ denotes the operation of dereferencing.

```
try_me_else C
  if( A > StackEnd ) goto Error
  CE := E; E := A - env_offset;
  CE(E) := CE; BP(E) := C;
  TR(E) := TR; H(E) := H;
  [ BP := C; ]
retry_me_else C
  BP(E) := C;
  [ BP := C; ]
trust_me
  BP(E) := Fail;
  [ BP := Fail; ]
dettry_me_else C
  if( A > StackEnd )
                          goto Error;
  CE := E; E := A - env_offset;
  CE(E) := CE;
  [ BP := C; ]
detretry_me_else C
  [ BP := C; ]
dettrust_me
  [ BP := Fail; ]
return
  C := CP(E);
  A := E + env_offset; E := CE(E);
  goto C;
execute C
  A := E + env_offset; E := CE(E);
  goto C;
```

```
save N
  A := E - (const + var_size * N)
get_atom_first An, Rn
  if( Rn <> Var(atom,An) ) {
   if ( Tag(Rn) <> ref ) goto BP;
   else {
    Ref(Rn) := Var( atom, An );
    Push( TRAIL, Ref(Rn) );
   }
  }
get_atom An,Rn
  if( Rn <> Var(atom,An) ) goto BP;
get_structure Sn,Rn
  if(( Tag( Rn ) <> ref) or
   Ref(Rn) <> Var(str,Sn) ) goto BP;
  else S := Value( Rn ) + var_size;
picktvar Rn
  Rn := Ref( S ); S := S + var_size;
movereg Rn,Rm
  Rm := Rn;
movereg_deref Rn,Rm
  Rm := Deref( Rn );
bldtval Rn
  Ref( S ) := Rn; S := S + var_size;
put_structure Sn,Rn
  Push( HEAP, Sn );
  Rn := Var( ref, H );
  S := H + var_size;
```

Let us consider the following simple program to illustrate the compilation process:
$reverse([], X) <= X;$
$reverse([X|Y], Z) <= reverse(Y, [X|Z]);$
$Goal : write(reverse([1,2,3,4], []));$

The global analysis gives the following information:
$< (free - by - value, ground, ground),$
$function, deterministic, no - choice - point,$
$(determenistic, no - choice - point, tail - recursion - is - impossible)$
$(determenistic, no - choice - point, tail - recursion - is - possible) >.$

All rules in the program are deterministic. So, there is no need for choice points (in FAM to set choice points, a special instruction *setcp* is used). The method of returning the value is *free-by-value*. The compiler produces the following intermediate FAM code for the *reverse*:

```
reverse:
   dettry_me_else reverse2
   get_atom [] R_1
   save 0
   movereg R_2 R_1
   return

reverse2:
   dettrust_me
   get_structure . R_1
   picktvar R_60
   picktvar R_1
   save 0
   movereg R_2 R_59
   put_structure . R_2
   bldtval R_60
   bldtval R_59
   execute reverse
```

7 More optimizations

In this section we consider some optimizations that do not yet implemented in the current version of the Flang compiler but (at least, the first of them) worthy to be implemented. There are so many optizations, so, it is not too easy to choose really serious ones. In this section we consider only two of them.

The first optimization is the optimization of *the last use of a structure*. Sometimes the system can prove that in some part of a program a structure is used for the last time. In this case, the system may use destructive assignment to change the structure itself instead of copying. In the *QuickSort* program (see section 2) the sorted list is used only once. So, the system can use the list itself to build the result. For this, it is necessary to add to the global analysis the module which recognizes the point of computation when the structure is built. The essential condition of applicability of this optimization is absence of choice points between the creation of the structure and the last use of it. This optimization is difficult for implementation but can improve a produced code drastically. The optimization sometimes allows to generate the code pecular to imperative languages, because it permits to use destructive assignment for the structure and not to spread the Heap. The example of a program when this optimization can be applied very efficiently is the *QuickSort* program. In the relation *partition* (see section 2) it allows to use destructively the list $[Y|Z]$ for creating new lists $W1$ and $W2$. Thus, the produced code becomes not less efficient than corresponding programs written in imperative languages with the use of the destructive assignment to modify lists (see the next section).

Another optimization is not so powerful but it gives the possibility to execute purely functional programs in the purely functional way. This optimization can be characterized as 'avoiding saving address of the father activation record $CE(E)$ in the Stack'. It can be illustrated in the folowing way. Let us remove the operation $CE(E) := E$ from instructions *try_me_else, dettry_me_else, etc.* and add it to the definition of the instruction *call.* In some situations the global analysis shows that during computation of some rule no choice points will be created. Then the distance between the activation record of the rule and calls in this rule is a constant computed at compilation. So, saving the address of the activation record is unnecessary. Unfortunately, this optimization is not compatible with the garbage collector implemented in the Flang system.

8 Estimation of the Flang compiler perfomance

We compared the performance of the Flang system with well known commercial Prolog systems and with Pascal programs developed and compiled in TurboPascal. We used the IBM PC XT computer with frequency 4.77 MHz. All but three benchmarks are described in section 2.

This work shows that the performance of the Flang system is very high. The compiled code is much faster not only than the code produced by not too effecient compiler of Arity/Prolog, but than the code produced by the TurboProlog, which language was 'spoiled' for the sake of efficiency by introduction of user-defined declarations of types. Our compiler does not have this kind of information given by the user. But it have the same (and much more!) information from the global analysis.

The comparison of compiled Flang code with corresponding programs written in TurboPascal shows that for some class of benchmarks when in both programs is implemented the same algorithm, the Flang program usually has about the same speed as the Pascal one. The main problem is that in Pascal some destructive tools (such as the assignment) are available. They admit quite different and extremely efficient strategies of computations. When these strategies are applied, the current version of the Flang compiler produces a slower code. But we are sure that the application of the optimization described in the previous section can improve the situation drastically.

The first benchmark is the algebraic version of the sorting algoritm:
$X.Y.Z <= X > Y -> Y.X.Z;$
The Prolog program having the same strategy of computation is
$algsort(X, [Y|Z], R) :- X > Y, !, algsort(X, Z, U), algsort(Y, U, R).$
$algsort(X, Y, [X|Y]).$

The usual list plays in this program the role of '.'. The corresponding Pascal program is recursive and uses just the same algorithm. The systems sorted the list [300, 299, 298, ..., 1].

The same list was sorted by *QuickSort* programs. The Pascal program essentially uses the advanatages of the imperative programming style, so the Flang program is about two times slower (a similar ratio holds for the other systems using the global analysis: *e.g.* see [5]). We need new optimizations (for instance, the one described in the previous section) to improve the situation.

Let us introduce other benchmarks. The first of them is the Flang program for the *Ackermann* function:
$ackermann(0, N) <=> N + 1;$
$ackermann(M, 0) <=> ackermann(M - 1, 1);$
$ackermann(N, M) <= ackermann(N - 1, ackermann(N, M - 1));$

The $fun6$ function is defined as follows:
$fun6(0) <=> 1;$
$fun6(X) <= fun6(X - 1), fun6(X - 1), fun6(X - 1),$
$\quad fun6(X - 1), fun6(X - 1), fun6(X - 1);$
We computed the value of $fun6(7)$.

The function $prime(N)$ returns the list of all prime numbers equal or less than N:
$rem(Num, X) <= Num < X - > Num;$
$rem(Num, X) <= rem(Num - X, X);$
$check(Pri, 1) <=> true;$
$check(Pri, Den) <= rem(Pri, Den) <> 0, check(Pri, Den - 1);$
$prime(1) <=> [];$
$prime(X) <= check(X, X - 1) - > [X|prime(X - 1)];$
$prime(X) <= prime(X - 1);$
The computed value is $prime(200)$.

The last function $backfun$ illustrates the usefulness of the information given by the global analysis of choice points:
$backfun(0) <=> true;$
$backfun(X) <= f(1, 1000) - > backfun(X - 1);$
$f(X, Dim) <= X > Dim;$
$f(X, Dim) <= f(X + 1, Dim);$
The computed value is $backfun(100)$.

benchmark	*Flang*	*Pascal*	*TurboProlog*	*Arity*
algsort	19.88	20.93	34.11	69.60
qsort	11.05	6.04	20.21	
ackermann	18.95	19.68	23.34	
fun6	25.35	29.48	35.37	201.55
prime	6.24		14.17	
naïve reverse	9.46		11.97	
backfun	8.9		31.17	

9 Conclusion

In the conclusion, we would like to summarize the main results of the paper: Programs written in the high level languages can be compiled into an extremely fast code. For the significant class of programs this code has about the same efficiency as programs written in imperative languages (C, Pascal, *etc.*). Quite powerful systems supporting high level languages can be developed even on relatevely small and weak computers.

Now we are developing a semi-commercial version of the Flang system. It contains not only the fast compiler we described above but also supports constraint satisfaction technique [3] and some other mechanisms interesting to theoretical considerations and useful in practice.

References

[1] A.Mantsivoda. *Flang: A Functional-Logic Language.* Proc. of Int. conf on Processing Declarative Knowledge.- Kaiserslautern, July 1991.

[2] D.H.D.Warren. *An Abstract Prolog Instruction Set.* Technical Note 309 SRI International, Menlo Park, CA, October 1983.

[3] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge, 1989.

[4] H.Boley. A relational/functional Language and its Compilation into the WAM. SEKI Report SR-90-05, University of Kaiserslautern, 1990.

[5] P.L.van Roy *Can Logic Programming Execute as Fast as Imperative Programming?* PhD Dissertation, University of California at Berkeley, November 1990.

[6] A. Taylor *High Performance Prolog Implementation.* PhD. Dissertation, Basser Department of Computer Science, University of Sydney, June 1991.

[7] D. Turner. *An Overview of Miranda.* SIGPLAN Notices, vol 21, No 12.

[8] G.Janssens, B. Demoen, A.Marten. *Improving the Register Allocation in WAM by Reordering Unification.* Proc. 5th Int. Conf. Symp. Logic Programming, pp.1388-1402, MIT Press, Cambridge, MA, 1988.