Generating Analyzers with PAG

Martin Alt
Florian Martin
Reinhard Wilhelm

Technischer Bericht **A10/95'**
Universität des Saarlandes
FB 14 Informatik
66041 Saarbrücken

December 16, 1995

Abstract

To produce high quality code, modern compilers use global optimization algorithms based on *abstract interpretation*. These algorithms are rather complex; their implementation is therefore a non–trivial task and error–prone. However, since they are based on a common theory, they have large similar parts. We conclude that analyzer writing better should be replaced with analyzer generation.

We present the tool PAG that has a high level functional input language to specify data flow analyses. It offers the specification of even recursive data structures and is therefore not limited to bit vector problems. PAG generates efficient analyzers which can be easily integrated in existing compilers. The analyzers are interprocedural, they can handle recursive procedures with local variables and higher order functions. PAG has successfully been tested by generating several analyzers (e.g. alias analysis, constant propagation, interval analysis) for an industrial quality ANSI-C and Fortran90 compiler. This technical report consists of two parts; the first introduces the generation system and the second evaluates generated analyzers with respect to their space and time consumption.
Keywords: data flow analysis, specification and generation of analyzers, lattice specification, abstract syntax specification, interprocedural analysis, compiler construction

Contents

1 The System **3**

1.1 Introduction . . . 3
1.2 Theoretical Background . . . 3
1.3 The System . . . 4
1.3.1 Overall Structure . . . 4
1.4 The Specification Language . . . 4
1.4.1 The Domain Specification Language . . . 5
1.4.2 The Data Flow Description Language . . . 6
1.4.3 Analyzer Description . . . 7
1.4.4 The Interface Description . . . 8
1.5 Interprocedural Analysis . . . 8
1.5.1 Fundamental Algorithms . . . 8
1.5.2 Our Approach . . . 8
1.5.3 Formal Description . . . 9
1.5.4 Mappings . . . 11
1.6 Practical Results . . . 12
1.7 PAG and Beyond . . . 13

2 Practical Evaluation **15**

2.1 Time and Space consumptions . . . 15
2.2 The Orderings . . . 15
2.3 Practical Measurement . . . 17
2.4 Conclusion and Further Work . . . 19

Chapter 1

The System

1.1 Introduction

Research in compiler generation has concentrated mostly on front end and lately on back end generation. The optimization phase has not received much attention. Only a few systems [27, 29, 33] for generation of analyzers were designed and built. All of them apply ad-hoc methods or heuristics if the language has subroutines. We present a new generative system for interprocedural analyses, PAG, that is able to produce analyzers which can be applied in several different compilers by instantiation of a well designed interface. The system is based on the theory of abstract interpretation. The philosophy of PAG is to support the designer of an analyzer by providing three languages for specifying the data flow problem, the abstract domains, and the compiler interface. This simplifies the construction process of the analyzers as well as the correctness proof and it results in a modular structure. The specifier is neither confronted with the implementation details of domain functionality nor with the traversal of the control flow graph or syntax tree nor with the implementation of suitable fixpoint algorithms. In the paper, we first briefly summarize the theory of the data flow analysis in section 1.2, then we introduce the structure of the generating system in 1.3. Section 1.4 discusses the specification languages, followed by the presentation of our interprocedural solution mechanism in section 1.5. Some measurements are presented in section 1.6. Finally, section 1.7 summarizes the related works and exhibits some possible extensions.

1.2 Theoretical Background

The data flow analysis practiced nowadays was introduced mainly by [17] and refined by [16]. It is based on a *control flow graph (CFG)* that contains a node for every statement or basic block in a procedure and an edge for a possible flow of control. Furthermore we add a unique entry node s and exit node e and a labelling function that yields a syntax tree fragment for every node. A *data flow analysis problem (DFP)* or *instance of a data flow analysis framework* is a combination of such a graph with a complete lattice of values, called the *underlying lattice*, and a family of functions (one for each node). These functions express the local semantics and are therefore called *transfer functions*. If every transfer function is monotonic the problem is called a *monotone problem*. If they are even distributive it is a *distributive problem*.
To describe the solution of a data flow problem we'll first define the semantics of a path $\pi = n_1, \ldots, n_k$ in the *CFG*:

$$[\![\pi]\!] = \begin{cases} id & \text{if } \pi = \epsilon^1 \\ [\![(n_2, \cdots, n_k)]\!] \circ [\![n_1]\!] & \text{otherwise} \end{cases}$$

The desired solution of the *DFP* is the union of the semantics of all paths applied to bottom, historically called the *meet over all paths solution*:

$$\mathbf{MOP}(n) = \bigsqcup \{[\![\pi]\!](\perp) \mid \pi \text{ is a path from } s \text{ to } n\}$$

for every node n of the *CFG*, where $\perp$ is the bottom element of the lattice. As the set of all paths from s to n is usually infinite, this solution is in general not computable. Therefore, the *minimal fixpoint solution*

[1] ϵ denotes the empty path

was introduced:

$$\mathbf{MFP}(n) = \begin{cases} [\![s]\!](\perp) & \text{if } n = s \\ [\![n]\!]\left(\bigsqcup\{\mathbf{MFP}(m) \mid m \text{ predecessor } n\}\right) & \text{otherwise} \end{cases}$$

Kam has proved in [16] that for every monotone data flow problem the **MFP** is greater (with respect to the ordering of the lattice) than the **MOP** solution, and therefore a save approximation. Moreover, if the *DFP* is distributive, both are equal. The interprocedural version of this theorem is presented in [18].

To solve a data flow problem, different iterative solution algorithms can be used which are guaranteed to terminate if the problem is monotone and the underlying lattice has no infinite ascending chains. Some of them, like the worklist algorithm, are presented in [14]. Other algorithms are the bounded fixpoint iteration from [23] or the higher order chaotic iteration sequences in [24]. If the chains in the lattice are infinite or if a speed up of the fixpoint operation is needed, widening and narrowing, explained in [7], is an appropriate method to solve the equation system.

1.3 The System

So let's recall what's needed for a data flow analyzer:

1. a complete lattice D;
2. a meta transfer function which, if applied to a node of a *CFG*, yields a (monotone) transfer function from D to D;
3. a fixpoint algorithm with the appropriate data structures, e.g. an iteration algorithm and a working list;
4. a control flow graph, which is the input for the analyzer.

Some of these parts do not change much from one analyzer to another. To shorten the time of implementation and to simplify maintenance of the resulting system, we designed a special purpose programming language to describe (and create) such analyzers. By freeing the compiler designer from routine matters, he can focus on the crucial details. Our support language needs three specification components: the underlying lattice, the transfer functions and the solution method. This idea is not new. There have been a couple of experiments of generating analyzers, but only a few of them were successful. On one hand, the specification language should be powerful enough to cover a large class of problems, on the other hand the generated code must be sufficiently efficient if it is intended to work in a real compiler and not only to be a prototype for performing some tests.

1.3.1 Overall Structure

PAG has been designed to generate analyzers for compilation systems. The basic assumption is that such a compilation system offers an interface to the syntax tree and the control flow graph of the program to be analyzed. This representation of the intermediate structures can be found in nearly any compiler.

The interface to the control flow graph has to offer functions to walk over the graph, to get the identifier of a node, and to fetch the syntax tree which is the label of the node. To decouple the implementation of data structures of the graph from their logical functionality, the compiler designer has to write an interface which can be used for all generated analyzers working with this compiler. It is necessary to know the structure of the abstract syntax tree. It can be best defined by an extended tree grammar. From this grammar, the appropriate access and walk functions can be generated. Therefore PAG takes as input a file which describes the structure of the possible trees. Furthermore there are input files that construct the underlying lattice, the transfer functions and global parameters of the analyzer like the used fixpoint algorithm. All these input files are described in detail in section 1.4. An overview of the different input files to PAG is shown in figure 1.1.

1.4 The Specification Language

A PAG specification is divided in four parts: one for the definition of data types in the language DATLA, another for the description of the syntax trees, a third to specify the main structure of the analyzer, and a last to define the transfer functions in a language called FULA. This whole description is compiled into

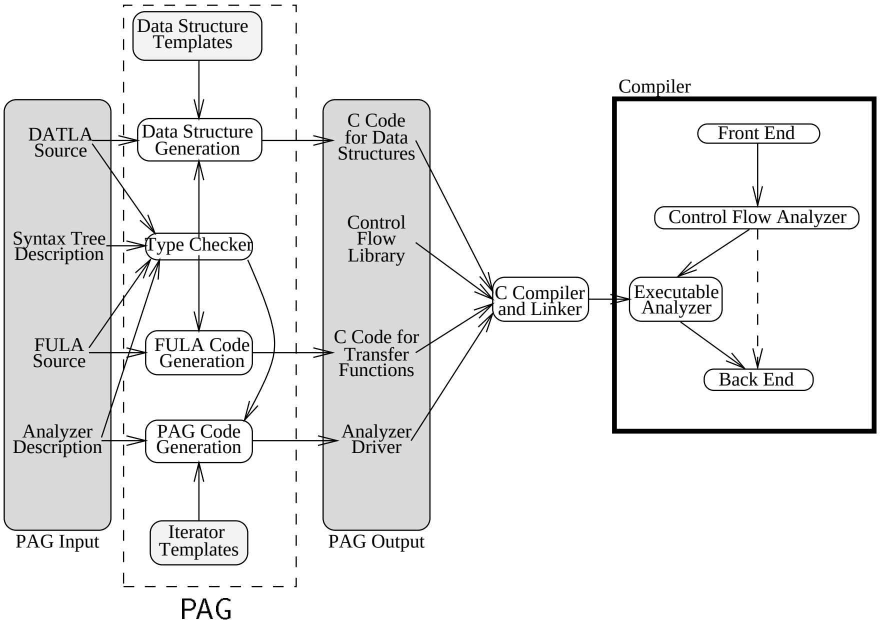

Figure 1.1: The Structure of PAG

ANSI C code that performs the analysis outlined in section 1.5. The structure of these four parts will be shown in detail now. First, we present some notational conventions. We use `typewriter` style for reserved words, and *italic* style for nonterminal symbols. Among these 'v' is used for variables, 'p' for patterns, 'e' for expressions in general and 'b' for boolean valued ones. Finally '$\{\}$' is used for grouping and 'a^+' and 'a^*' for the usual repetitions. A complete example of an analyzer specification is given in appendix 2.4. It is the description of an analyzer to detect assignments to live variables.

1.4.1 The Domain Specification Language

It is used to define the data types and data manipulation routines for the analyzer. Therefore the language is called DATLA, an acronym for data type definition language. We distinguish between sets and domains (complete lattices). They are constructed bottom up. This means that simple sets like numbers or enumerated sets build the base for more complicated sets like the set of all functions from numbers to truth values, which can be used as input for other set generating operators again. So we have some basic sets, and set operators. In addition there are operators to construct domains from sets, and to combine domains to new domains.

The specification is divided into two subparts: one for the set definitions starting with the keyword `SET` and the other for the domain definitions with the keyword `DOMAIN`. Each definition is an equation with a single name that has to be defined on the left side and an operator applied to a couple of names on the right side. The predefined sets are up to now: `snum`: the set of signed integers, `unum`: the set of unsigned integers, `real`: the set of all floating point numbers, `bool`: the set of truth values `true` and `false`, `chr`: the character set (ASCII), and `string`: the set of all character sequences. This list can easily be extended even with user-defined types. These must have implementations of every function the interface contains. Another possibility to generate a basic set is to enumerate a finite number of elements. Complex sets can be formed out of the basic ones through the following operators:

1. disjoint union of a finite number of sets: Disjointness is obtained by tagging the element of the sets participating in the union.
2. construction of the tuple space of a finite number of sets
3. power sets: building the set of all subsets of the original set
4. set of lists of elements of a set
5. set of functions from a set S_1 to a set S_2

In the second part of the specification, domains can be formed in one of the following ways:

1. enumeration of elements and of a (complete) partial order. The system checks whether it forms a lattice.
2. flattening of a set S: S is transformed into a domain by adding a least element $\perp$ and a greatest element $\top$. The elements of S are pointwise incomparable.
3. lifting of a domain D: new top and bottom elements are added which are greater respectively smaller than all other elements. The ordering of D is preserved.
4. building the power domain of a set or domain: the generated ordering is set inclusion.
5. construction of the tuple space of a finite number of domains. Tuples are ordered component–wise.
6. building the domain of functions from a set into a domain with point–wise ordering.

In contrast to other generators, PAG allows simultaneously recursive definition of sets and domains. In this way, for instance, the tree type can be easily expressed. But this and other features in DATLA result in the fact that infinite sets and domains, even with infinite chains, can be defined. Therefore the user has to make sure that only a finite part of the domain is used, or has to guarantee the termination of the analysis in another way. Environments are an example where only a finite part of some infinite domain is used. It is typical for program analysis that at each program point, some kind of information is stored for every variable that occurs in the program. This is usually modelled by a function from variables to the needed information. Obviously this domain has infinite chains because the set of variables is infinite. But in every program only a finite subset of variables is used, and so termination is guaranteed.

1.4.2 The Data Flow Description Language

The main part of the description of an analyzer is the specification of the data flow functions. For every node in the control flow graph, there has to be a function that transforms an incoming flow value into an outgoing one. This is expressed in a functional language FULA which was designed especially for that purpose. FULA programs can be compiled to efficient C code.

Overview: FULA is a first order functional language with eager evaluation. It has static scoping rules and the user defined types from DATLA. Interestingly, the language does not provide an explicit fixpoint operator. As we will see, the absence of an embedded fixpoint operator and the eager evaluation semantics do not restrict significantly the language constructs that PAG is able to analyze. In FULA every expression has a unique type that can be derived statically by a type inference algorithm. There are no implicit type casts in FULA. Any change of the type must be made explicit. For occurrences of a variable the corresponding definition is the syntactically innermost definition for that variable. Binding constructs are function definitions, `case` and `let` expressions.

Transfer Functions: The whole FULA source is split in two parts: one for definitions of auxiliary functions and one for the transfer functions. These transfer functions are written in a special notation: they don't need a name and are defined via patterns matching the labels of the control flow graph. They have an implicit parameter named `@` which is the incoming data flow value and they have to return a data flow value again.

Functions: There are two different types of functions in FULA. Firstly the functions defined in the language itself, and secondly those declared in DATLA. The latter are seen as datatypes and can be arguments to the first sort. This distinction is made because of the first order character of the language. So it is possible to write a (FULA) function, that takes a (DATLA) function as argument. For example a function that modifies the value of an environment for a given variable.

Functions can only be defined with a name, which means that there is no lambda expression. Definitions are made by using patterns, and the cases can be spread over the whole specification. A single case looks like $f(p^*) = e$. For each function there can be an additional type definition. Nested functions are not allowed in FULA.

Patterns: They are used to define functions and in `case` expressions. Only *linear* patterns are usable, which means that the same variable is allowed only once per pattern. Pattern expressions may be nested. The following patterns are defined, while some static semantical restrictions, like type conditions, are to be followed:

1. constants: these can be elements of predefined sets like integers as well as user defined constants of enumerated sets or domains. Two special constants are $\perp$ and $\top$, the bottom and top elements of all lattices.

2. variables, but each variable once per nested pattern.
3. empty list and empty set. There are two types of empty lists: those defined in DATLA and those introduced by the syntax tree (see 1.4.4). They are notated as `[]` or `[!]` and {} for the empty set.
4. cons patterns for both types of lists: p`:`p.
5. tuples: `(`p`,`$\cdots$`)`. Notice that a tuple pattern matches always if the sub patterns match.
6. wild card: it is denoted by an underbar _ and matches every input.

Expressions can be:

1. constants (built-in and enumerated)
2. if expressions: **if** b **then** e_1 **else** e_2 **endif** for conditionals. This can be seen as the only non eager construct in DATLA, because e is evaluated first, and then e_1 or e_2 but never both.
3. let expressions: `let` $\{\ v_i = e_i;\}^+$ `in` e to introduce a number of variables v_i local to the expression e. The eagerness results in the fact that every e_i is evaluated before e is evaluated.
4. case expressions: `case` $\{v_j\}^+$ `of` $\{\{p_j\}_i^+$ => $e_i;\ \}^+$ `endcase` gives the possibility to examine the structure of one or more expressions bound to the variables v_j. The result of this expression is the value of the first e_i for which all p_j's match the values of the v_j's
5. function application: both kinds of functions DATLA and FULA can be applied to zero or more expressions. The number of expressions applied to must correspond to the arity of the function because functional expressions that would result if a function is applied to more or less arguments than defined, are not allowed in a first order language.
6. print expressions: `print` (e_0) `++` e is equivalent to the expression e with the side effect that e_0 is printed out.
7. built in function application (pre- and infix notation). These are functions that are generated for certain data types. All these functions are listed below (for a more refined documentation see [21]):
   - for every type: equal `=` and not equal `!=`
   - for every domain additionally: `lub`, `glb`, $\sqsubseteq$, $\sqsupseteq$, $\sqsubset$, $\sqsupset$
   - `unum`, `snum` ,`real`: mathematical operations like: plus `+`, minus `-`, times `*`, integer division `/`, modulo `%`, amount $|\,.\,|$; bitwise operations: and `&`, or `|`, xor `^`; comparison functions: greater >, greater equal >=, less <, less equal <=; cast functions: `snum`, `unum`, `chr`, `real`
   - `bool`: and `&&`, or `||`, not `!`
   - `chr`: cast functions `snum`, `unum`, `real`
   - `string`: concatenation `.`, character selection [.]
   - flatted, lifted domains: `lift`, `drop`
   - lists: cons `:`
   - sums: `is-`*type_name*, `down-`*type_name*, `up-`*type_name* to lift an element of a sum component into the sum and vice versa.
   - tuples: select `!`, tuple construction $(e_1, \cdots, e_n)$
   - power sets: add an element `~`, subtract an element `#`, member test `??`
   - functions: application .{ . }, value changing .[. \ .], creation with default element [− > .]

1.4.3 Analyzer Description

In order to generate analyzers, some additional declarations are necessary:

1. `direction`: values can be `forward` or `backward`. This specifies whether the data flow is along the edges of the control flow graph or in the opposite direction.
2. `carrier`: the name of the domain that is used for the analysis; i.e. the type of the flow values.
3. `combine`: the name of a FULA function, that merges information which comes over different edges. This is usually the function `lub` for the least upper bound.
4. `init`: the initial value that is associated with every control flow node. Usually the neutral element of the combine function ($\perp$).
5. `init_start`: an initialization value for the start node (the end node for a backward problem).

1.4.4 The Interface Description

This part of the specification is used to define the structure of the syntax tree which is constructed from the source program by the compiler. With this definition, the PAG compiler is able to generate access functions for the tree, following certain naming conventions. This is needed because the generated analyzer must be able to walk over the syntax tree to determine for instance the instruction given on a control flow node. The form of the tree is introduced as a tree grammar with two additions. Firstly, there is a notation to introduce lists of nonterminals in order to gain more simplicity in notation and handling. Secondly we have a notation *nonterminal* == *simple_type* to identify a class of nonterminals with a built in type. For such a rule, PAG generates functions to cast an element of *nonterminal* into an element of *simple_type*. Generally the nonterminals of the syntax tree and lists of them are considered as types that are usable in FULA. The concrete syntax is quite obvious, and is skipped here with a pointer to the PAG reference manual [21].

Another part of the interface is that to the control flow graph. But this interface is quite simple and fixed irrespective of the programming language. So the control flow graph is abstracted by a library that has to be supported by the compiler. It includes functions to fetch the successors or ancestors of a node, or to access the corresponding syntax tree fragment.

1.5 Interprocedural Analysis

In the previous sections, we have focused our interest on intraprocedural analysis. Yet good programming style requires a lot of small procedures that are used at many program points. So it is a must not only to look inside every procedure on its own but at the whole program in order to achieve good analysis results. As already mentioned, the number of problems with interprocedural analysis is large: to achieve excellent results very much time is needed, and fast algorithms do not find out everything possible. So one has to find a compromise between time consumption and precision.

1.5.1 Fundamental Algorithms

There are different methods to handle procedures in data flow analysis. We shortly summarize some techniques:

- Non–recursive procedures can be inlined, and the intraprocedural algorithm can be applied. This is only applicable for small programs because the transformed program may grow exponentially.

- Procedure calls may be considered as ordinary statements that make all available information invalid. Or destroy only all information based on those objects that may be modified by that call; this assumes a previous analysis phase that computes for each procedure the set of potentially modified variables (data flow problem over call graph).

- Use a two-level algorithm: first compute an abstract form of the procedure (called *effect* or *jump* in [12]), which maps flow information from the beginning of a procedure to its end; then do standard iteration and use the related effect function at any procedure call; the computation of the effect functions brings up new restrictions on the underlying lattice, as they cannot be computed in general or even if they can, the representation may grow exponentially ([4]) This method is especially useful if the lattices are finite. Many restricted versions of the constant propagation relay on that, e.g. constant copy propagation [19] or even demand-driven versions [9].

- Do an analysis within the procedure body: one variant of this method summarizes the effects of all calls at the beginning of a procedure and another keeps different calls separate. This approach is also known as *call string* approach and is described in [25].

1.5.2 Our Approach

Our approach tries to achieve the following goals:

- to guarantee termination for every input and every analyzer that terminates intraprocedurally;
- to keep the additional effort of the compiler designer for the interprocedural analysis low;
- time and space complexity should be as low as possible to guarantee the practical usability of the generated analyzer;

- the results from the interprocedural analysis should be as good as possible.

To get a precise solution (this means the **MOP** solution) for the interprocedural case for all problems that are terminating intraprocedurally is not possible. As stated in [25] this is only possible if the domain is finite or the DFP is distributive or if there is no recursion in the program. All three cases restrict our mentioned goals too much. The approach we have chosen is similar to the call string method, because it offers a flexible technique that can niftily trade time and space for precision (see section 1.5.4). One can observe that in most 'every day live' programs calls to a procedure from different call sites[2] are made with distinct values. And in addition to that, a large number of function calls are non–recursive. So the different call sites are worth the effort to analyze them separately. Furthermore it can be useful to keep deeper levels of calls separated. This will be clarified by the following example:

```
static int i;                    r() {                    q() {
main() {                            q();                     i += 1;
   i = 1;                        }                        }
   r();
   r();
}
```

If we separate every call chain of length up to two, in a constant propagation analyzer (compare [30]) we can figure out that the call of `q` in `r` is made twice with different values. The separation of the calls is achieved by introducing arrays or vectors of data flow values for each node of the control flow graph instead of using only one value per node. In the different array elements we can store the data that comes from different call sites. If there are dynamically more calls than there are fields in the vector (which has a static length) they will be merged together with the (specified) combination operation.

After the analysis has finished, an extra advantage of this vector is that it can be used in different ways: for a simple optimization inside the procedure the meet of all elements can be used because that is the flow value which is valid for all incarnations of the procedure. But it can also be used to determine if it is useful to specialize the procedure for certain values.

1.5.3 Formal Description

Definition 1 (Program Representation)
We represent a program P with the procedures $P_0, \ldots, P_n$, where P_0 is the main procedure as a **super graph** $G^* = (N^*, E^*, s_0, e_0)$. *$G^*$ consists of a number of intraprocedural control flow graphs $G_0, \ldots, G_n$ which represent the procedures of P. In difference to the standard CFGs, every call in G^* is represented by three nodes: a call site node, a return site node, and a local node. The call node has one edge to the entry node of the called procedure and another to the local node. The return node has the exit node of the called procedure and the local node as a predecessor.*

Figure 1.2 is an example of a super graph. The local node between call and return node is useful to model the behavior of local variables: the compiler designer can specify that after the local node the flow value is constantly bottom[3]. Then the implicitly performed combine operation before the return node will always yield the value returned from the procedure. But in the constant propagation for example it is also possible to set all global variables at the local node and all variables local to the call at the procedure exit to bottom. Then the local variables arrive at the return node directly from the call and all other variables are coming from the procedure exit. This is correct if none of the local variables can be modified (due to aliasing) in the procedure. Therefore the concept of the local node results in a large variety of possibilities to handle local values for the specifier of the analyzer; the interface to the abstract syntax tree allows even to access information that has been computed by other compiler parts; e.g. alias graphs.

There are two problems with the construction of a super graph: if a call is made to a function variable the called procedure is not known. Then, there must be an edge to the entry of every (potentially called) procedure. The second one arises if the called procedure is not known because it's not in the source code like any kind of library function. Then we introduce a dummy procedure with which the unknown call is connected. The analyzer designer has to specify worst case assumptions for that dummy procedure, or can

[2] i.e. different places in the source code
[3] if the combine function is specified as a meet

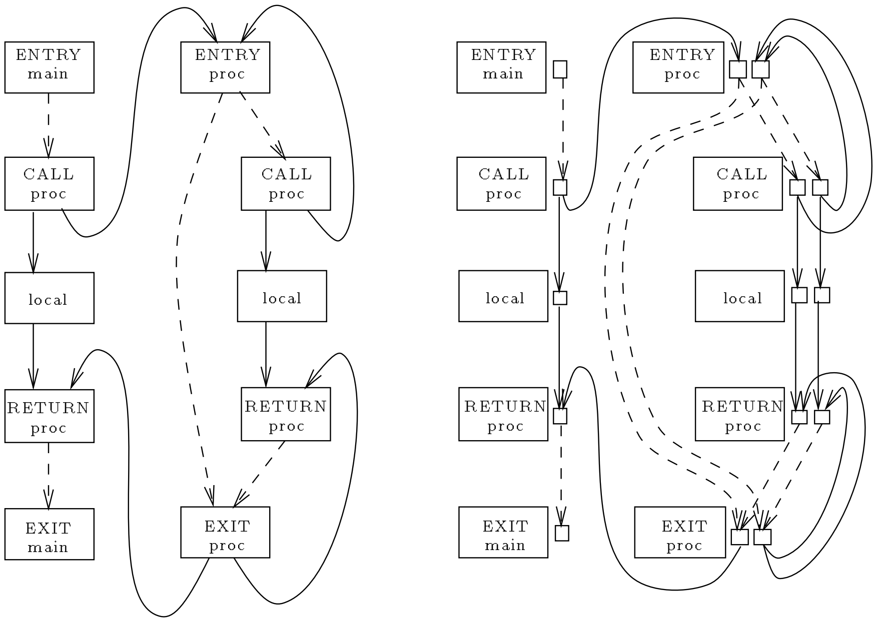

Figure 1.2: A super graph and its extended super graph

do better if he knows the behavior of the library function. Thus, PAG supports unlike all other generators a specification mechanism for the semantics of libraries.

In a super graph we can carry out a standard intraprocedural data flow analysis. So we don't have any additional expenses for computing the effect of the functions. As a refinement we should use only interprocedural valid paths in the graph. A path is called a *valid interprocedural path* if it contains only matching pairs of call and return nodes of a procedure like a correctly braced expression. Exact definitions can be found in [19, 20].

Now we can introduce the discrimination of the data of different call sites by assigning not only a single value of the lattice with every control flow node, but an array of values. It is clear that the length of these arrays in every procedure should be the same for each of its control flow nodes, and at least one.

Definition 2 (Graph Extension)
For a super graph G^*, we define the triple $(G^*, \mathbf{Arity}, \mathbf{map}_c)$ to be a graph extension if **Arity** *is a function that maps every node $n \in N^*$ to a natural number, the length of the data flow array, and $\mathbf{map}_c$ is a family of functions for every call site $c = call\ P$ where $\mathbf{map}_c : [1..\mathbf{Arity}(c)] \rightarrow [1..\mathbf{Arity}(entry\ P)]$ maps every position of the data flow array of the calling procedure to a position in the array of P.*

Instead of considering paths between nodes of the super graph, we use now paths between the elements of the data flow arrays. So a new graph with pairs of nodes and vector positions as nodes can be defined.

Definition 3 (Extended Super Graph)
For a graph extension $(G^*, \mathbf{Arity}, \mathbf{map}_c)$ we call the graph $G_E^* = (N_E^*, E_E^*, s_E^*, e_E^*)$ extended super graph, with $N_E^* = \{(n,i) \mid n \in N^* \text{ and } i \in [1..\mathbf{Arity}(n)]\}$, $s_E^* = (s^*, 1)$, $e_E^* = (e^*, 1)$, and $((n_1, i_1), (n_2, i_2)) \in E_E^*$ iff n_2 is an entry node and $i_2 = \mathbf{map}_{n_1}(i_1)$, or n_1 is an exit node and $i_1 = \mathbf{map}_c(i_2)$ with c being the call belonging to n_2, or $i_2 = i_1$.

An extended super graph with its extended paths is shown in figure 1.2. The interprocedural analysis can be performed by applying an intraprocedural fixpoint algorithm to the extended super graph. The merging of the data at different call sites is done automatically, if the $\mathbf{map}_c$ are chosen appropriately: In figure 1.2, the two edges of the call to proc from itself are leading into the same data flow element at the head of proc. So a meet of the two values of the call site will be calculated if the second element of the vector at the entry of proc is needed. The corresponding element from the exit node will be duplicated and propagated to the return node inside proc. To solve a data flow problem with extended graphs, it is necessary to find suitable map and arity functions. With these it is possible to tune the analysis: the higher the arities the better the precision we can achieve, but the more time and space is needed.

1.5.4 Mappings

The task to select a mapping is mainly to find a compromise between time/space complexity and preciseness of the analysis. So we will now explain some methods to calculate pairs of arity and mapping.

1. In the simplest case, the arity of each procedure is one, and the mapping functions are the identity. So the information of each call to a procedure is mixed together.

2. Another simple way is to count the number of calls to a procedure p in the program text (this is the number of control flow edges in the super graph that are leading to the entry of p) and to take this number as the arity of p. Of course one must choose the arity of P_0 as one although it is formally not called. Then the $\mathbf{map}_c$ functions project all elements of the tuple at the call site to a single fixed position in the array of the called procedure.
   The effect of these functions is that the meet of the data flow information at every call site in the program is kept separately in the called procedure p, but if there are further calls from p to some other procedure q the information is mixed up with the flow values of all other calls to p. In practice, it has shown that the arity of some procedures calculated with this method becomes quite large in real programs (up to 100). So it seems to be reasonable not to use higher values for the arities in large programs, due to space and time restrictions.

3. To be more precise, it is necessary to keep deeper levels of the call separated. If there are for example two different call sites c_1 and c_2 of a procedure p and p calls q we would like to have as many fields in the array of q as there are fields at c_1 plus the number of fields that are at c_2 in which we can map the resulting information of c_1 and c_2. So the arity of a procedure should be the sum of the arities of all call sites. As induction base we have $\mathbf{Arity}(\text{entry } P_0) = 1$. But this works only if the call graph is acyclic which means that the program is non–recursive.

   So we have to find the strongly connected components (SCC, see [22]) in the call graph G and consider them as a single procedure in a call graph G'. To this acyclic call graph G' we can apply the method described for the non–recursive case, with the additional rule that the arity of a SCC is the arity calculated for the non–recursive case multiplied by a constant k which reflects the fact that there can be many recursive calls inside the SCC. Afterwards, we expand the melted nodes again and assign to every procedure the arity of the compound node.
   The resulting mapping for calls leading to a procedure outside any SCC or calls from outside into a SCC is simple because every element in the call site vector has a corresponding one in the vector at the procedure entry. For the other mappings, different ways can be chosen. An example is shown in figure 1.3.

4. If the underlying lattice D is finite a mapping can be constructed that results in the precise solution of the DFP. Call chains of length up to $|D|^2 * K$ (K is the number of call sites in the program) must be analyzed separately.

Summary

So let's shortly summarize the advantages and disadvantages of our approach:

— we are usually not precise, but this can be achieved by using an appropriate mapping if the domain is finite.

— complex lattices and large programs result in analyzers that use a lot of memory and time; to overcome the space problems, we added a garbage collector for the flow elements (FULA is functional and therefore has a copy semantics).

\+ every intraprocedural *DFP* can be extended to an interprocedural analyzer without further restrictions of the domain.

\+ additional specification is only needed for parameters, return values and local variables.

\+ tuning is possible through the choice of arity and mappings.

\+ the method yields detailed results about the different calls.

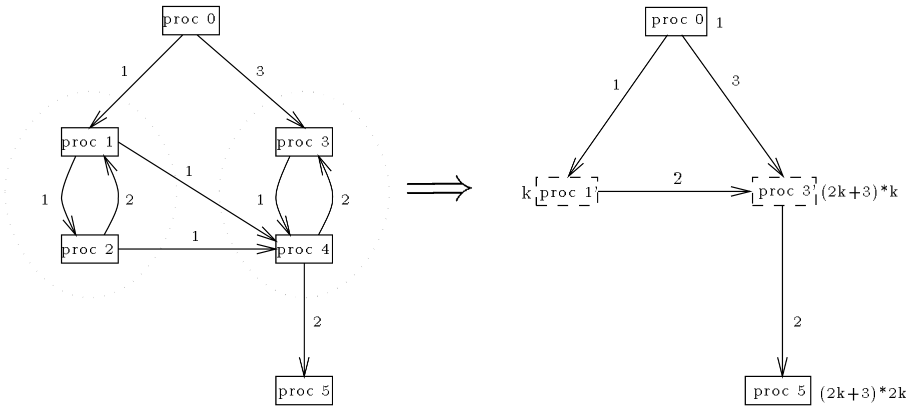

Figure 1.3: The calculation of the arity in method 3. The numbers on the edges mean the numbers of call sites, and the numbers at the nodes are the calculated arities, e.g. the edge `proc 2` to `proc 1` means that there are two call sites of `proc 1` in `proc 2`.

	Lines of Specification				Generated	Interface
	DATLA	FULA	syntax	$\sum$	Code (kB)	(lines)
copy constant propagation	233	12	144	389	268	1283
linear constant propagation	400	14	144	558	309	1251
full constant propagation	628	12	144	784	371	1283
alias analysis	203	10	144	357	215	1084
live variables	95	10	144	249	222	1054
dominator analysis	22	3	144	169	147	1084

Figure 1.4: The size of specification and the generated analyzer

1.6 Practical Results

PAG generated analyzers have successfully been tested in a large compiler system with ANSI-C and Fortran90 front ends as well as with different back ends (including SPARC) in the **ESPRIT** project COMPARE. We have specified the following data flow problems:

- copy constant propagation: only assignments of the form `x := c` where `c` is a constant, and `x := y` are taken into account.
- linear constant propagation: here additionally, statements of the form `x := c*y + d` are considered, where `c` and `d` are constants.
- full constant propagation: every right side is taken into account. Note that the lattice for this analyzer is infinite and the transfer functions are not distributive.
- alias analysis: computes *must* and *may* aliases
- liveness: a well known bit vector problem
- dominator: similar to liveness

These analyzers run together with handwritten and generated compiler phases (called *engines*) in the **CoSy** compilation model that has been introduced by [1]. In figure 1.4 we have listed the size of the different specifications and analyzers. The specification of the syntax tree is the same for all analyzers as well as a large part of the hand coded interface listed in the last row. As there are some differences in the subsequent treatment of the computed data, they are not fully identical. We measured the practical relevance of the 2. mapping of section 1.5.4 by applying the full and the copy constant propagation to several ‘real’ programs. The results are shown in figure 1.5. We have also generated analyzers for a completely different compiler which takes a subset of Pascal as input.

program	*procs*	*nodes*	*objects*	IPC_C	IPC_F	CON_C	CON_F	$\#obj_C$	$\#obj_F$	$\#s_C$	$\#s_F$
`cdecl`	32	3209	835	7	7	1010	1010	2/80	3/80	357	357
`ed`	46	3448	784	7	9	1481	1509	7/59	1159	138	138
`fft`	7	638	195	3	3	439	439	0/19	0/19	62	62
`flex`	128	11974	2301	49	49	4926	4980	59/355	59/336	1072	1072
`flops`	2	640	139	9	9	5820	6209	90/47	161/61	240	1105
`gzip`	99	8332	1740	11	11	4353	4358	36/250	37/280	1148	1148
`heap`	3	361	94	8	8	1432	1432	13/34	13/34	55	55
`linpack`	12	1196	278	6	6	587	610	28/51	32/55	490	490
`ratfor`	51	3200	762	0	0	934	934	6/54	6/54	323	323
`twig`	81	4525	1085	6	6	1332	1332	3/64	3/64	375	375
`xmodem`	30	4460	1157	38	38	4849	4963	6/164	12/166	520	526
`clinpack`	12	1257	285	6	6	668	691	28/51	32/55	490	490
`cloops`	126	3212	620	127	131	13341	14436	56/177	60/179	1377	1388
`dhry`	13	662	239	8	8	279	286	2/13	3/14	99	99
`search`	3	89	34	2	2	189	189	7/8	7/8	9	9
`whet`	7	492	121	3	3	397	448	16/44	40/49	104	104

Practical results measured for the copy constant propagation (X_C) and the full constant propagation(X_F).

IPC: the number of constant objects at the procedure headers
$\sum_{entry\ p} |\{o | (\sqcup_{i=1}^{Arity(p)} flow(p)[i])(o) \neq \bot, \top\}|$

CON: the sum of the number of constant object at all nodes
$\sum_{flownode\ n} |\{o | (\sqcup_{i=1}^{Arity(n)} flow(n)[i])(o) \neq \bot, \top\}|$

#obj: x/y, x the number of constant source code variables,
y the number of constant temporaries (frontend introduced)

#s: foldings of subexpressions

Figure 1.5: Practical Measurements

1.7 PAG and Beyond

As far as we know, only a few analyzer generators have been implemented. Many papers have discussed analyzer generators from a theoretical point of view, but not many serious efforts have been undertaken to produce practical implementations. Projects we know about are the following:

1. Tjiang and Hennessy [27] presented an analyzer generator *Sharlit* which uses *path compression* for evaluating the path functions. The user has to give simplification rules and has to support the implementation of the flow values and graph routines in low level C code. There is no possibility of automatic generation of the domain code and no automatic handling of procedures.

2. Venkatesch and Fischer [29] presented a tool *Spare* which is based on the abstract interpretation framework. It is developed mainly for testing purpose of some abstract interpretations. Moreover it has no built in facilities to handle a function mechanism.

3. Yi and Harrison [33] developed an abstract interpretation based analyzer generator. It has the ability to tune the analysis by using a restricted form of narrowing. But it allows only domains of finite cardinality.

We have shown in this paper that PAG is able to generate analyzers that can analyze large programs. Specifying an analysis like the copy propagation shortens the time of implementation to something between one and two hours if the interface is already there (which is needed only once per compilation system). One of the authors was able to adapt an already existing specification of constant propagation to include alias information within three man hours.

For the future we plan to investigate additional things:

- new fixpoint algorithms: we plan to implement some other fixpoint algorithms and investigate a hybrid approach, where we can mix different algorithms and keep the correctness and termination properties.

- reduction of the space consumption: the vectors of the data flow elements will be large, if the program mainly consists of procedure calls. In that case most of the entries at procedure headers are the same; thus, a melting of these elements may result in less precise data flow information, however, with decreased time and space consumption. This can be generalized by replacing the equality with a distance notion. This distance can be inductively defined on the structure of the lattices. The user can specify a concrete distance d and during the analysis elements e_1, e_2, within this distance, are replaced by the result of the (specified) combination function $combine(e_1, e_2)$. Then the mappings of section 1.5.4 are dynamically redefined.

- automatically reasoning over PAG specifications: an interesting research direction is the analysis of PAG specifications. Using an eager first-order functional language for specifying the transfer functions allows automatic proof of some properties like monotonicity.

Chapter 2

Practical Evaluation

2.1 Time and Space consumptions

There exists few knowledge about the space and time consumption of global data flow analyzers on real world programs as well as the accuracy of the results. The combination of fixpoint algorithm, abstract domain, transfer functions, and call string length has big effects on the practical usability. Here we present the results of applying some of the different combinations of these parameters to realistic programs.

The main challenge in building an interprocedural data flow analyzer is the exponential space and time consumption for specific programs that has been proven in theory. However, sometimes full analysis is needed, independently of the costs. In [2] the data flow analyzer generation system PAG has been introduced. It generates efficient interprocedural data flow analyzers from high level specifications for compilation systems that are flow based. PAG allows to specify the abstract domains, the transfer functions and the interface for control flow graph access. The fixpoint algorithm is currently implemented using a worklist approach (also known as single step technique). The worklist contains the set of nodes that have to be processed at least once. Then a node is selected and processed (related transfer function is applied). If the data flow value changes, its successors are inserted in the worklist (for a forward problem). The efficiency of the analyzers depends on a smart selection of the next node, which has been discussed extensively in the past for the intraprocedural case ([11, 13]). The most general and common idea is that nodes are ordered and the worklist implementation selects the minimal or maximal node. This technique is called *priority queue*. For some control flow graphs an optimal ordering can be computed e.g. the reducible flow graphs ([14]).

PAG generated analyzers have to handle control flow graphs which are often irreducible; that arises from the presence of procedures (higher order), non-local jumps and exceptions. We have investigated the effect of different orderings on the costs of interprocedural analyzers. We specified and generated a conditional constant propagator ([31]) for ANSI-C and applied it to realistic programs (table 2.1).

2.2 The Orderings

We implemented a bottom-up fixpoint algorithm that is based on a worklist containing those control flow nodes whose data flow values have to be recomputed. In opposition to other implementations, where the members of the priority queue are control flow nodes, we use pairs of nodes and indices to represent extended super graph nodes. We implemented the worklist using a priority queue. The priorities of the control flow nodes are given by seven different heuristics.

The optimal ordering is computable, but assumes an explicit representation of the extended super graph which is too large for use in practise. But the extended super graph differs only at the procedure entry and exit points from the super graph and can be safely approximated by it. Thus we base our heuristics only on the structure of the super graph.

In the remainder of this section we describe the different orderings of the control flow nodes. We have implemented seven different orderings of the control flow nodes. They differ in the treatment of edge types and the visiting order of successors. The first four orderings do not take the types of edges into account i.e. they handle procedure call/return edges like ordinary flow edges.

dfs: depth first order

the nodes are ordered according to a *depth first search* visiting sequence of the control flow nodes;

bfs: breath first search

the nodes are ordered according to a *breath first search* visiting sequence of the control flow nodes;

scc$_d$: strongly connected components (scc)

it first computes the strongly connected components of the super graph. They are ordered by a topological ordering ([28]). The nodes of the sccs are ordered by the **dfs** strategy;

scc$_b$: strongly connected components

it first computes the strongly connected components of the super graph. They are ordered by a topological ordering. The nodes of the sccs are ordered using the **bfs** strategy;

hyp$_d$:

it first computes the strongly connected components of the super graph. They are ordered by a variant of a topological ordering (ATS). The nodes of the sccs are ordered using the **dfs** strategy;

hyp$_b$:

it first computes the strongly connected components of the super graph. They are ordered by a variant of a topological ordering (ATS). The nodes of the sccs are ordered using the **bfs** strategy;

In addition we used the *chaotic ordering* (**cha**) which means, that the worklist is implemented as a stack.

We now present an algorithm for the computation of a topological ordering of sccs. Because the sccs are computed intraprocedurally, the projection of the interprocedural graph to these sccs may be cyclic; we start from the set of sccs and order them as long as there are minimal elements; if none exists we have to apply a heuristic to determine which cycle we want to break up and where. Our heuristic is to choose a scc that has a minimal number of predecessor (control flow) nodes which belong to sccs that are already ordered.

Algorithm ATS:

The algorithm ATS orders the strongly connected components of a control flow graph. The root (R_e) of a scc e is the node which is reached first by a dfs algorithm. The algorithm is related to that of [3], because it defines a *weak topological ordering.*

Input: strongly connected components of a flow graph $\{s_1, \ldots, s_m\}$

Output: total ordering *ord* of sccs

$SCC := \{s_1, \ldots, s_m\}$; count:=1; $\forall\ i \in \{1, \ldots, m\}$ $ord(s_i) := 0$;
while $SCC \neq \emptyset$ do
if $\exists\ s \in SCC$ with no predecessor then $sel := s$;
else
    select $\{y_1, \ldots, y_n\} \subset SCC$ with minimal $po(R_{y_i})$ and $on\_cycle(R_{y_i}) = true$
    if $\exists\ i$ with y_i is procedure begin node then $sel := y_i$; else
    if $\exists\ i$ with y_i is procedure end node then $sel := y_i$; else
    $sel = y_1$; endif
    $ord(sel)$:=count++;endif
$SCC := SCC - \{sel\}$
enddo

The function *po* (predecessors ordered) is defined as :

$$po(x) = |\{z \mid ord(R_z) \neq 0 \wedge \exists\ edge\ z \rightarrow x', R_{x'} = R_x\}|$$

It computes the size of the set of control flow nodes that belong to sccs which are predecessors of x. A small value indicates that this node can be selected next. A better strategy is not to select the node with minimal number of unordered predecessors but the node with the largest fraction of already ordered number and the total number of predecessors. This assumes a more sophisticated arithmetic and is not efficient enough. The predicate *on_cycle* determines whether a node belongs to a cycle of the control flow graph; this is justified because cycle should be analyzed completely before leaving it.

`program`	description	lines	procedures	flow nodes	objects	table
fft	fast fourier trans.	418	8	400	195	2.2
flex	scanner generator	5985	129	9209	2301	2.12
ed	editor	1506	81	2854	782	2.7
bison	parser generator	6438	155	13109	3562	2.11
heap	heapsort	546	4	240	94	2.5
twig	code generator	2092	81	3576	1085	2.13
gzip*	compactor	4056	100	6840	1740	2.6
xmodem	communication	2060	31	3526	1157	2.10
whetstone	benchmark	508	8	349	121	2.9
linpack	benchmark	821	13	904	285	2.3
cdecl	part of C++ compiler	2831	33	2536	835	2.8
find*	unix command	7341	211	10190	2704	2.4

Table 2.1: Testsuite

2.3 Practical Measurement

The testsuite consists of a rather fair set of *real* programs of reasonable size. We listed in table 2.1 the name of the program, the number of source code lines, the number of procedures, the number of control flow nodes (not of the extended super graph), the number of objects (superset of variables) and the number of the table with the related results. The star * means that this program contains higher order functions. The conditional constant propagator consists of 798 lines of analyzer specification, 15 lines of lattice specification and 144 lines of interface specification. The PAG generated code is 617kB ANSI-C.

We investigated all combinations of the seven orderings and mappings with call string length zero and one and compared the results in time and preciseness. The tables have the following structure:

Structure of the Tables

The first row contains the name of the program whereas the second shows the preciseness in terms of available constants (constant objects, which are not necessarily used), interprocedural constants (ipc, constants at procedure headers) and foldings of form $y(x)$; it means that x source code variables are replaced by their constant values, and y foldings are done caused by the former replacements; i.e.

ipc:	the number of constant objects at the procedure headers
	$\sum_{entry\ p} \lvert\{o \mid (\sqcup_{i=0}^{Arity(p)} flow(p)[i])(o) \neq \bot^{\S}, \top\}\rvert$
available:	the sum of the number of constant objects at all nodes
	$\sum_{flownode\ n} \lvert\{o \mid (\sqcup_{i=0}^{Arity(n)} flow(n)[i])(o) \neq \bot, \top\}\rvert$

We listed for each program the steps of the control (how often a node is selected), the time for the analysis in seconds and the average size of the priority queue during analysis. The winner of each competition is marked with a $\oplus$, and the loser with a $\ominus$. The following table contains the results for specific programs, where an enlarged call string length results in higher preciseness.

§ we use the lattice **variables** $\rightarrow$ **flat**(num++real++string)

Program	call string length	available	ipc	foldings		steps	time	work	strategy
ed	2	1362	10	10	(11)	20451	55.2	56	$\mathbf{hyp}_b$
xmodem	2	4109	38	12	(12)	56571	228.9	150	$\mathbf{hyp}_d$
whetstone	2	935	7	91	(54)	4858	6.1	11	$\mathbf{hyp}_d$
linpack	2	416	6	60	(21)	8625	7.7	6	$\mathbf{hyp}_d$
cdecl	2	797	7	4	(2)	68180	53.2	190	$\mathbf{hyp}_b$
	3	803	7	4	(2)	63453	99.7	111	$\mathbf{hyp}_b$
flex	2	3812	50	22	(59)	110052	942.1	233	$\mathbf{hyp}_b$
	3	8861	124	24	(84)	169980	1806.0	356	$\mathbf{hyp}_b$
	4	8867	129	24	(84)	284739	4892.7	373	$\mathbf{hyp}_b$
heap	2	1101	8	8	(15)	1616	2.6	6	$\mathbf{hyp}_b$

Results:
For most programs the strategy **cha** (chaotic iteration) behaves worst whereas $\mathbf{hyp}_b$ is the fastest one. Increasing the call string length from 0 to 1 gives always more available constants, but does not necessarily result in code improvement because they are not used (table 2.10). Available interprocedural constants are found in most of the programs.

Observation 1
The selection best ordering of the control flow nodes depends on the program and the call string length.

We can see the correctness of this theorem by observing that the runtime of the analyzer consists of two parts: the time inside the control (priority queue, binomial heaps, straightforward list) and the time inside the abstract functions. A very fast control which selects unfortunately more often nodes with expensive abstract functions may be less efficient than a very sophisticated control which selects expensive node less often. Thus, the combination which cooperates best has to be found.

Observation 2
The analysis time is **not** *a monotonic function in terms of precision.*

Assume the abstract functions do follow the ascending chains in the lattice step by step; the earlier this process stops, the faster and more precise are the analysis results. This does not mean that shorter analysis yields always better results.

Observation 3
The preciseness is **not** *a strictly monotonic function in terms of the call string length even not before stabilisation.*

This means that the preciseness can be the same for two lengths of the call string but increase again for higher values. The implication is bad: if we don't get more preciseness by increasing the call string length, we can not claim that this is also true for all larger call strings. That is clear from the theoretical point of view, but we have found the practical confirmation.

We now discuss the table for each program in detail; for this we will use *csl* as short cut for call string length and *ats* as short cut for the two **ats** strategies hyp_d and hyp_b:

fft(table 2.2): The number of available constants and ipcs increases from *csl* 0 to 1 but unfortunately no code optimization can be done. The *bfs* strategies are better than the *dfs* based ones for this program. *csl* larger than 1 does not give additional constants.

linpack(table 2.3): The number of foldings increases from *csl* 0 to 1 and the analysis time decreases for all except the chaotic iteration. The size of the worklist increases with the call string length.*bfs* strategies behave worse than *dfs* based ones.

find(table 2.4): The precision increases with the *csl* and more foldings can be done. The analysis times are high due to the use of higher order functions.

heap(table 2.5): *cha* has best runtimes which is very surprising. The *dfs* strategies behave generally worse than its *bfs* counterparts. For *csl* 1 the 'expected' results are obtained, the *ats* strategies win. The precision does not grow for *csl* bigger than 1.

gzip(table 2.6): The analysis times look very bad for gzip but that arises from intensive use of higher order functions. This is probably also the reason why no interprocedural constants are found at all.

ed(table 2.7): The precision increases from *csl* 0 to 2, but *csl* 2 finds only more available constants. Although different *ats* strategies win, the distance to others is not significant.

cdecl(table 2.8): The precision increases for *csl* 0 to 3. *bfs* strategies behave worse than *dfs* ones. For *csl* 1 the runtime is very long. The *ats* methods can analyze it in reasonable time.

whetstone(table 2.9): The precision increases from *csl* 0 to 2; here we have the interesting fact that the foldings are equal for *csl* 0 and 1, but increase significantly for *csl* 2. The analysis time decreases for *csl* 2 and one gets more precise information in short time.

xmodem(table 2.10): The precision and the number of foldings increase from *csl* 0 to 2. *csl* 2 finds a lot of interprocedural constants, but needs also more analysis time.

bison(table 2.11): The precision increases with the *csl*, but does not result in more foldings. The analysis times are relative close compared to the other programs.

flex(table 2.12): The precision increases from *csl* 0 to 4 and results in more foldings for *csl* $\leq$ 3. *bfs* strategies are better than its *dfs* counterparts, but only the *ats* are suited for practical use.

twig(table 2.13): The precision increases remarkably from *csl* 0 to 1.

2.4 Conclusion and Further Work

We have presented the results of applying a (generated) conditional constant propagator which also keeps track of reference parameter and function results to a set of real-world programs. Our analyzer is based on the *call string approach* ([25]). It separates different context information for different calls up to a fixed depth. We implemented a bottom-up[§] fixpoint algorithm using a priority queue, where the priorities are computed according to an ordering of the control flow node. The classical ordering like *depth first search* ([26]), *intervals analysis* ([11, 13]) or *chaotic iteration strategies* ([3]) fail either in the interprocedural setting, are only applicable to reducible flow graph or assume further properties like absence of non-local gotos or deal with slightly different topics like the search of nearly optimal widening points. We developed a new method (*ats*) which has been proven practically relevant.

We currently implement the *functional approach* of [25] and plan to compare it with our first technique. Additionally, we plan to generate parallel analyzers ([10]).

[§] due to non-local gotos, which make top-down algorithms very complicated

fft							
avail:118 ipc:0 fol:0(0)				avail:273 ipc:3 fol:0(0)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha	1874	3.1	12	cha $\ominus$	12750	39.8	10
bfs	2657	2.9	14	bfs	5373	9.2	14
dfs $\ominus$	4169	6.9	23	dfs	14782	29.8	27
scc$_d$	2080	2.5	21	scc$_b$ $\oplus$	5096	8.6	22
scc$_b$	2057	2.2	15	scc$_d$	15015	32.9	40
hyp$_d$	1423	1.1	10	hyp$_d$	7778	11.0	9
hyp$_b$ $\oplus$	1082	1.0	10	hyp$_b$ $\oplus$	6488	8.9	7

Table 2.2: Results for `fft`

linpack							
avail:262 ipc:6 fol:39(8)				avail:400 ipc:6 fol:60(21)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha	3581	8.1	19	cha	14164	24.9	97
bfs $\ominus$	17278	1003.1	79	bfs	20186	64.8	102
dfs	1906	6.2	43	dfs	6421	6.2	67
scc$_d$ $\oplus$	1905	6.1	43	scc$_d$	6420	6.3	68
scc$_b$	17122	995.5	81	scc$_b$ $\ominus$	20148	66.2	103
hyp$_d$	1483	10.5	54	hyp$_d$ $\oplus$	4498	4.0	44
hyp$_b$	1528	10.7	54	hyp$_b$	4755	4.7	45

Table 2.3: Results for `linpack`

find							
avail:4910 ipc:9 fol:42(27)				avail:5925 ipc:17 fol:42(29)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha	33064	11768	285	cha $\ominus$	???	>6000	???
bfs	29294	90.9	405	bfs	192770	2548.0	487
dfs $\ominus$	38725	144.9	450	dfs	193822	5365.7	445
scc$_d$	35247	143.4	469	scc$_d$	170254	716.0	576
scc$_b$ $\oplus$	26022	79.2	468	scc$_b$	171163	817.6	689
hyp$_d$	33810	93.1	172	hyp$_d$ $\oplus$	130136	350.4	285
hyp$_b$	33868	98.5	156	hyp$_b$	181888	1312.0	339

Table 2.4: Results for `find`

heap							
avail:341 ipc:2 fol:2(8)				avail:940 ipc:8 fol:7(13)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha $\oplus$	1117	1.5	8	cha $\ominus$	4880	8.4	12
bfs	1633	2.4	240	bfs	3785	7.9	18
dfs $\ominus$	2858	4.8	18	dfs	3729	7.7	21
scc$_d$	2817	4.8	18	scc$_b$	3704	7.9	21
scc$_b$	1599	2.3	17	scc$_d$	3755	7.8	20
hyp$_d$	2034	2.6	12	hyp$_d$	2957	5.9	15
hyp$_b$	1940	2.5	12	hyp$_b$ $\oplus$	2949	5.8	14

Table 2.5: Results for `heap`

gzip							
avail:2221 ipc:0 fol:16(34)				avail:2976 ipc:0 fol:24(36)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha	20617	103.6	213	cha $\ominus$	???	>2600	???
bfs	20424	396.5	380	bfs	122710	1043.2	967
dfs	18432	198.7	309	dfs $\ominus$	172217	2510.7	862
scc$_d$	18428	214.5	309	scc$_d$	171363	2608.1	864
scc$_b$ $\ominus$	20396	398.5	380	scc$_b$	122688	1100.1	969
hyp$_d$	11403	43.0	256	hyp$_d$ $\oplus$	81376	435.3	526
hyp$_b$ $\oplus$	11161	39.4	258	hyp$_b$	82344	446.6	523

Table 2.6: Results for `gzip`

ed							
avail:998 ipc:7 fol:8(8)				avail:1331 ipc:9 fol:10(11)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha $\ominus$	7885	13.4	55	cha $\ominus$	31331	101.5	83
bfs	6180	8.0	81	bfs	22145	42.2	63
dfs	7379	9.4	140	dfs	23181	46.0	109
scc$_d$	7342	11.1	139	scc$_d$	22045	50.1	119
scc$_b$	6338	8.2	88	scc$_b$	21855	44.1	73
hyp$_d$	5618	7.7	58	hyp$_d$ $\oplus$	11894	16.5	37
hyp$_b$ $\oplus$	5163	6.0	59	hyp$_b$	11753	17.2	35

Table 2.7: Results for `ed`

cdecl							
avail:506 ipc:2 fol:2(1)				avail:789 ipc:7 fol:4(2)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha $\ominus$	6004	14.11	160	cha	184997	19989.5	471
bfs	6277	8.8	182	bfs $\ominus$	???	> 37545	???
dfs	4860	5.3	182	dfs	24256	28.2	418
scc$_d$	4808	4.8	181	scc$_d$	23933	28.8	430
scc$_b$	4355	4.3	137	scc$_b$	266353	28164.0	441
hyp$_d$	3377	3.0	175	hyp$_d$ $\oplus$	14822	13.7	370
hyp$_b$ $\oplus$	3379	2.8	185	hyp$_b$	15155	13.7	360

Table 2.8: Results for `cdecl`

whetstone							
avail:255 ipc:3 fol:79(40)				avail:266 ipc:3 fol:79(40)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha	1332	0.6	12	cha $\oplus$	11941	6.2	57
bfs	3077	3.3	39	bfs $\ominus$	15960	72.7	73
dfs $\ominus$	4444	12.1	51	dfs	9170	11.0	57
scc$_d$	4415	12.1	50	scc$_d$	9129	11.3	58
scc$_b$	3025	3.6	39	scc$_b$	15738	66.8	75
hyp$_d$	769	0.5	30	hyp$_d$	4942	21.6	35
hyp$_b$ $\oplus$	764	0.5	50	hyp$_b$	4906	21.1	35

Table 2.9: Results for `whetstone`

xmodem							
avail:998 ipc:7 fol:8(8)				avail:1331 ipc:9 fol:10(11)			
call string length 0				call string length 1			
`order`	`steps`	`time`	`size`	`order`	`steps`	`time`	`size`
cha $\ominus$	11407	108.5	161	cha $\ominus$	???	>300	???
bfs	11273	27.7	373	bfs	40871	66.8	265
dfs	9552	19.5	316	dfs	52278	220.5	576
scc$_d$	7596	17.8	309	scc$_d$	36999	113.5	507
scc$_b$	6101	9.8	217	scc$_b$	40820	82.8	274
hyp$_d$	6020	11.5	241	hyp$_d$ $\oplus$	26200	64.6	218
hyp$_b$ $\oplus$	5108	9.7	238	hyp$_b$	26222	80.5	227

Table 2.10: Results for `xmodem`

bison

avail:2258 ipc:11 fol:10(5) — call string length 0

order	steps	time	size
cha ⊖	29503	85.5	470
bfs	25424	83.2	692
dfs	27446	80.5	655
scc_d	26343	67.0	635
scc_b	22643	73.2	585
hyp_d	21273	54.3	406
hyp_b ⊕	20955	54.3	408

avail:2899 ipc:12 fol:10(5) — call string length 1

order	steps	time	size
cha ⊖	155852	728.5	508
bfs	106794	262.2	624
dfs	134463	453.3	671
scc_d	135435	363.8	502
scc_b	104238	272.1	734
hyp_d	93779	234.8	292
hyp_b ⊕	82156	184.1	279

Table 2.11: Results for `bison`

flex

avail:3285 ipc:45 fol:19(57) — call string length 0

order	steps	time	size
cha ⊖	24587	31.9	338
bfs	17809	29.6	427
dfs	19942	30.7	644
scc_d	19682	28.7	641
scc_b	17740	28.3	440
hyp_d	14304	19.6	363
hyp_b ⊕	14159	19.6	367

avail:3637 ipc:49 fol:22(59) — call string length 1

order	steps	time	size
cha	156312	564.5	525
bfs	113433	1552.8	849
dfs ⊖	84897	3320.2	953
scc_d	128415	3191.8	938
scc_b	89221	253.5	829
hyp_d	50398	101.1	387
hyp_b ⊕	48448	98.5	389

Table 2.12: Results for `flex`

twig

avail:599 ipc:0 fol:3(2) — call string length 0

order	steps	time	size
cha	8564	20.5	105
bfs	17099	105.0	359
dfs	7398	33.5	226
scc_d	7415	33.3	224
scc_b ⊖	25303	171.5	382
hyp_d	6017	17.2	143
hyp_b ⊕	5933	16.9	147

avail:3637 ipc:49 fol:22(59) — call string length 1

order	steps	time	size
cha	55865	91.0	460
bfs	37258	73.0	278
dfs	65588	165.0	438
scc_d ⊖	65557	174.2	467
scc_b	43506	132.2	429
hyp_d	33599	49.5	155
hyp_b ⊕	31212	46.6	162

Table 2.13: Results for `twig`

Bibliography

[1] Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy Compiler Phase Embedding with the Cosy Compiler Model. In *6th International Conference for Compiler Construction in Edinburgh*, volume LNCS786. Springer, 1994.

[2] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *SAS'95, Static Analysis*, LNCS 983, pages 33–50. Springer, 1995.

[3] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Conference on Formal Methods in Programming and their Applications*, number 735 in LNCS, pages 128–141. Springer Verlag, 1993.

[4] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *SIGPLAN Notices*, 21(7):152–161, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[6] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, January 1979.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992.

[8] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. *SIGPLAN Notices*, 27(7):212–223, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[9] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, California, January 1995.

[10] Yong fong Lee, Thomas J. Marlowe, and Barbara G. Ryder. Performing data flow analysis in parallel. In *Proceedings of Supercomputing '90*, pages 942–951, New York, November 1990.

[11] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 22–34, Palo Alto, California, January 1975.

[12] Dan Grove and Linda Torczon. Interprocedural Constant Propagation: A Study of Jump Function Implementations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, 1993.

[13] M. S. Hecht and J. D. Ullman. Flowgraph reducability. *SIAM Journal on Computing*, 1:188–202, June 1972.

[14] M.S. Hecht. *Flow Analysis of Computer Programs.* North Holland, New York, 1977.

[15] Neil D. Jones and Steven S. Muchnick. Even simple programs are hard to analyze. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 106–118, Palo Alto, California, January 1975.

[16] J.B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[17] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973.

[18] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, pages 125–140. Springer-Verlag LNCS 641, 1992.

[19] T. Reps M. Sagiv and S. Horwitz. Precise interprocedural dataflow analysis with application to constant propagation. In *TAPSOFT'95, Arhus, Denmark*, LNCS. Springer-Verlag, 1995.

[20] Florian Martin. Die Generierung von Datenflußanalysatoren. Master's thesis, Universität des Saarlandes, 1995.

[21] Florian Martin. *PAG Reference Manual.* Universität des Saarlandes, 1995.

[22] Kurt Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* Springer Verlag, 1984. ISBN 3-540-13641-X.

[23] Hanne Riis Nielson and Flemming Nielson. Bounded fixed point iteration. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–82, Albequerque, New Mexico, January 1992.

[24] Mads Rosendahl. Higher-order chaotic iteration sequences. In *PLILP'93, Tallinn, Estonia*, number 714 in LNCS, pages 332–345. Springer-Verlag, 1993.

[25] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[26] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[27] Steven W. K. Tjiang and John L. Hennessy. Sharlit – A tool for building optimizers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 82–93, San Francisco, CA USA, [7] 1992. ACM Press , New York, NY , USA. Published as SIGPLAN Notices, volume 27, number 7.

[28] Toda. On the complexity of topological sorting. *Information Processing Letters*, 35, 1990.

[29] G.V. Venkatesch and Charles N. Fischer. Spare: A development environment For Program Analysis Algorithms. In *IEEE Transactions on Software Engineering*, volume 18, 1992.

[30] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, New Orleans, Louisiana, January 1985.

[31] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[32] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* International Computer Science Series. Addison–Wesley, 1995.

[33] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, Charleston, South Carolina, January 1993.

Appendix A: Specification of Liveness

```
DOMAIN
varset = set(unum)
vars   = lift(varset)

PROBLEM livevar
direction : backward
carrier   : vars
init      : bot
init_start: lift(bot)
combine   : comb

TRANSFER

Assign(e1,e2):
minus(@,def(e1)) lub use(e1) lub use(e2)

Evaluate(exp):
@ lub use(exp)

If(exp,t,f):
@ lub use(exp)

FuncCall(_,params,_):
@ lub use_list(params)

Call(_,params):
@ lub use_list(params)

default:
@


SUPPORT

comb(a,b) = a lub b;

// Subtracts a list of unums from a set of unums
minus:: vars, [unum] -> vars;
minus(l_set,list) =
 let x <= l_set;
 in lift( case list of
               a:as => x # a;
                 [] => x;
          endcase);

// Calculate the used objects of expressions
use_list([!]) = lift({});
use_list(x::xs) = use(x) lub use_list(xs);

// Calculate the used objects of an expression
use::EXPR -> vars;
use(Content(ObjectAddr(obj))) = lift({} ~ id(obj));
use(Content(exp)) = lift(top);
use(Subscript(exp1,exp2)) = use(exp1) lub use(exp2);
use(Member(exp,_)) = use(exp);
use(Convert(exp,_)) = use(exp);
use(Cast(exp)) = use(exp);
use(Abs(exp)) = use(exp);
use(Neg(exp)) = use(exp);
use(Not(exp)) = use(exp);
use(Plus(exp1,exp2,_)) = use(exp1) lub use(exp2);
use(Diff(exp1,exp2,_)) = use(exp1) lub use(exp2);
use(Mult(exp1,exp2,_)) = use(exp1) lub use(exp2);
use(And(exp1,exp2,_)) = use(exp1) lub use(exp2);
use(Or(exp1,exp2,_)) = use(exp1) lub use(exp2);
use(Xor(exp1,exp2,_)) = use(exp1) lub use(exp2);
use(Div(_,exp1,exp2,_)) = use(exp1) lub use(exp2);
use(Quo(_,exp1,exp2,_)) = use(exp1) lub use(exp2);
use(Mod(_,exp1,exp2,_)) = use(exp1) lub use(exp2);
use(Rem(_,exp1,exp2,_)) = use(exp1) lub use(exp2);
use(_) = lift({});

// calculate a list of the defined objects in an expression
def(ObjectAddr(obj)) = id(obj):[];
def(_) = [];

// Get the identifier of an object
id(obj) = unum(val-INT(
  case obj of
    DataGlobal(id,_) => id;
    Local(id,_,_) => id;
    Parameter(id,_,_) => id;
    Register(id,_,_) => id;
    _ => error("Unknown object");
  endcase));


SYNTAX

START : STMT;

STMT: Evaluate(Expr:EXPR)
  | BeginProcedure(Params:Object*,Locals:Object*)
  | Assign(Lhs:EXPR,Rhs:EXPR)
  | Call(Proc:EXPR,Params:EXPR*)
  | FuncCall(Proc:EXPR,Params:EXPR*,Res:EXPR)
  | CallRet(Params:Object*,Locals:Object*)
  | EndProcedure(Params:Object*,Locals:Object*)
  | Goto(Target:EXPR)
  | Return(Value:EXPR,Next:EXPR)
  | If(Cond:EXPR,Then:EXPR,Else:EXPR)
  | EndFuncCall(Res:EXPR)
  | EndCall();

EXPR: NoExpr()
  | IntConst(Value:UNIV_INT)
  | RealConst(Value:UNIV_REAL)
  | BoolConst(Value:BOOL)
  | ObjectAddr(Obj:Object)
  | Content(Addr:EXPR)
  | Subscript(Base:EXPR,Index:EXPR)
  | Member(Base:EXPR,Field:Object*)
  | Convert(Value:EXPR,Rounding:ROUNDING)
  | Cast(Value:EXPR)
  | Abs(Value:EXPR)
  | Neg(Value:EXPR)
  | Not(Value:EXPR)
  | Plus(Left:EXPR,Right:EXPR,Strict:BOOL)
  | Diff(Left:EXPR,Right:EXPR,Strict:BOOL)
  | Mult(Left:EXPR,Right:EXPR,Strict:BOOL)
  | And(Left:EXPR,Right:EXPR,Strict:BOOL)
  | Or(Left:EXPR,Right:EXPR,Strict:BOOL)
  | Xor(Left:EXPR,Right:EXPR,Strict:BOOL)
  | Div(OnZero:EXPR,Left:EXPR,Right:EXPR,Strict:BOOL)
  | Quo(OnZero:EXPR,Left:EXPR,Right:EXPR,Strict:BOOL)
  | Mod(OnZero:EXPR,Left:EXPR,Right:EXPR,Strict:BOOL)
  | Rem(OnZero:EXPR,Left:EXPR,Right:EXPR,Strict:BOOL);

ROUNDING: Truncation()
  | Nearest()
  | Floor()
  | Ceiling();

Object: DataGlobal(cpid:INT,IsVolatile:BOOL)
  | Local(cpid:INT,IsVolatile:BOOL,Procedure:Object)
  | Parameter(cpid:INT,IsVolatile:BOOL,ParamKind:ParamKIND)
  | Register(cpid:INT,IsVolatile:BOOL,RegisterId:INT);

Section: Section(Sname:NAME);

ParamKIND: ByValue()
  | ByReference()
  | ByCopyInOut();

BOOL == bool;
NAME==str;
INT==snum;
UNIV_INT == snum;
UNIV_REAL == real;
UNIV_ADDRESS == unum;
```