Practical Comparison of *call string* and *functional* Approach in Data Flow Analysis*

Martin Alt   Florian Martin

Universität des Saarlandes, P.O. Box 151150, 66041 Saarbrücken,
{alt|florian}@cs.uni-sb.de

June 10, 1997

Abstract

The techniques which are used to implement (non-trivial) interprocedural data flow analyzers can be generally divided into two subsets: the *call string* and the *functional* approach as presented in [15]. Both differ in their time and space complexity as well as in the precision due to properties of the abstract domains and transfer functions. We have developed a data flow analyzer generator PAG which is able to produce interprocedural analyzers for both techniques. We specified two variants of constant propagation working in an ANSI-C compiler; a *copy constant* propagation that uses *distributive* transfer function and can be solved precisely, even interprocedurally [12], and a *full constant* propagator which includes an interpreter for expressions of the language. We present the practical relevant results applying both analyzers to a rather fair set of real-world programs and compare the space/time consumption of the analyzers versus their precision.

1 Introduction

The need for interprocedural analyses results from a conflict between the goals of software engineering and performance [11]. The usage of procedural and functional abstraction generalizes the code and makes it more extensible and maintainable. Optimizing compilers try to compute information about program points that can be used for code improvements. Intraprocedural optimization assumes worst case (means loss of all information) at procedure boundaries, whereas interprocedural optimizing compilers propagate information across procedure/function boundaries or eliminates the calls altogether by inline expansions for non-recursive calls. There exist two different approaches for the interprocedural analysis which have been discussed in [15]: the *call string* and the *functional* approach. The *call string* approach separates call chains (and the related data flow information) to a subroutine up to a fixed length. The precision increases with larger call strings. The maximal useful length for non-recursive programs is the height of the call DAG. For recursive programs the number of possible call strings is infinite. If there are no recursive procedures the number of call strings can be exponential in the number of procedures. Thus, we have to limit the call string length in any practical implementation to a certain length K.

The functional approach, on the other side, computes an abstract function for each subroutine. These are then used in a standard (intraprocedural-like) algorithm. However, the computation of the abstract functions may not terminate for lattices of infinite cardinality, even if they are of bounded height.

We designed and implemented an interprocedural data flow analyzer generator PAG, that is able to produce analyzers that use either a fixed call string length or the functional approach. We specified and generated two analyzers, included them in a (production quality) ANSI-C compiler, and measured the efficiency and precision for practical relevant applications.

By creating PAG we hope to close the gap between the theory of data flow frameworks and practical useful analyzers.

*Partially funded by **SFB** 124

The paper is organized as follows: first we briefly introduce the framework of dataflow analysis, then we discuss the two approaches and give practical measurements by applying these technologies to two different data flow problems. Finally, we draw a conclusion.

2 Techniques

The data flow analysis practiced nowadays was introduced mainly by [9] and refined by [8]. It is based on a *control flow graph (CFG=(V,E))* that contains a node for every statement or basic block in a procedure and an edge for a possible flow of control. The set of vertices V is a disjoint union of the set of nodes V_i for each subroutine p_i in the program*. Furthermore we add unique entry nodes s_i and exit nodes e_i for each procedure p_i. A *data flow analysis problem (DFP)* is a combination of such a graph with a complete lattice of values, called the *underlying lattice* ($\mathcal{L}$), and a family of functions (one for each node). These functions express the local semantics and are called *transfer functions* ($[\![\bullet]\!] : \mathcal{L} \to \mathcal{L}$). If every transfer function is monotonic (distributive) the problem is called a *monotone (distributive) problem*. To describe the solution of a data flow problem we need the semantics of a path $\pi = n_1, \ldots, n_k$ in the *CFG*:

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

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

$$\mathbf{MOP}(n) = \bigsqcup \{[\![\pi]\!](\perp) \mid \pi \text{ is a path from } s_0 \text{ to } n\}$$

for every $n \in V$. 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* was introduced:

$$\mathbf{MFP}(n) = \begin{cases} [\![n]\!](\perp) & \text{if } n = s_0 \\ [\![n]\!]\,(\bigsqcup \{\mathbf{MFP}(m) \mid m \text{ predecessor } n\}) & \text{otherwise} \end{cases}$$

If $\mathbf{MFP}(n) = \mathbf{MOP}(n)$ for all n, then the **MFP** solution is called *precise*. Kam has proved in [8] that for every monotone data flow problem the **MFP** is greater (with respect to the ordering of the lattice) than the **MOP** solution, and is therefore a safe approximation. Moreover, if the *DFP* is distributive the **MFP** solution is precise. The interprocedural version of this theorem is presented in [10].

2.1 Call String Approach

This approach considers procedure calls and returns as ordinary transfers of control, but it avoids propagation along interprocedurally invalid paths. This can be implemented by tagging the propagated data with a call string, which is a sequence of call nodes, whose calls are still unfinished. As the number of call strings may be infinite we bound their length to a number K and get an approximate solution by melting the information belonging to call strings with the same K-suffix. For a more practical implementation we use a vector of data flow elements. So we will separate the data flow information for all call strings up to length K. If we have to append a call site c to a string $\gamma = c_1 c_2 \ldots c_K$ which already has length K we throw away the first element c_1 of γ which results in $c_2 \ldots c_K c$. We transform this method into a vector based implementation by encoding the call strings as numbers which will correspond to positions in the data flow array: we assign to every call site c a unique number between one and $M-1$, where M is the number of call sites in the given program plus one. Then a call string $\gamma = c_{i_1} \ldots c_{i_n}$ corresponds to a K digit number m to base the M ($m = i_1 \ldots i_n$). To convert this number to the decimal system we choose the usual way: we multiply the n-th digit by M^n and add this for all digits ($m' = \sum_{1 \le j \le n} i_j * M^j$). So we get a number between zero and M^K where zero denotes the empty call string ϵ. Fortunately not all of these call strings are valid for each procedure. So we can omit all invalid numbers for a fixed procedure and squeeze the array, so that every position is used.

*furthermore, we assume the existence of a main routine p_0

$^{\dagger}\epsilon$ denotes the empty path

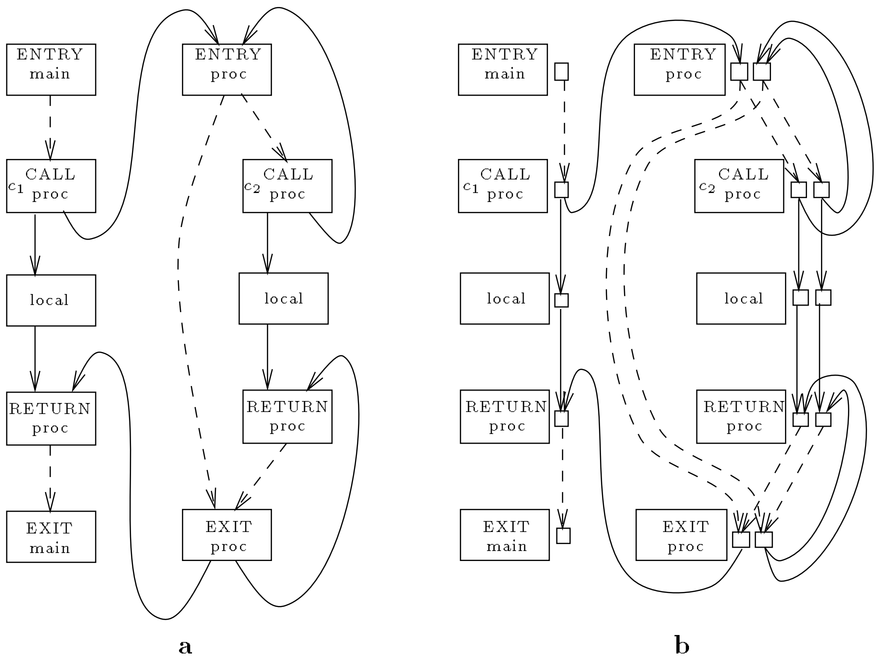

Figure 1: Example of the vector based callstring approach with K= 1

As an example we can consider the flow graph from figure 1a with $K = 1$. Valid call strings for `proc` are $c_1, c_1c_2, c_1c_2c_2, \ldots$. If we cut them to length one we have c_1, c_2. For `main` the empty call string is the only valid call string. So in the procedure `main` position zero of the data flow array of length three is used and in procedure `proc` the positions one and two are used. This results in the super graph in figure 1b.

If the underlying lattice $\mathcal{L}$ is finite a vector implementation can be constructed that results in the precise solution of the *DFP*. Call chains of length up to $|\mathcal{L}|^2 * M$ must be separated. This approach is related to [7].

2.2 Functional Approach

The functional approach computes for every procedure in the program a function which maps the data flow value which enters the procedure to the data flow value which leaves the procedure. Because the procedures can be simultaneously recursive it is necessary to do this computation for all procedures at once. In the general case for arbitrary lattices the only way to represent a function $\mathcal{L} \rightarrow \mathcal{L}$ is a table which tabulates input/output pairs. So it becomes clear that this table is guaranteed to stay finite if the lattice is of finite cardinality. As an optimization only those input/output pairs are computed for each procedure where the input value actually occurs.

Now we give a more formal description of this method: For each procedure p_i and each node $n \in V_i$ we define an element $\Phi(s_i, n) : \mathcal{L} \rightarrow \mathcal{L}$ which describes the effects to elements of $\mathcal{L}$ when they are propagated from the start s_i of procedure p_i to n. These functions are defined by:

$$\begin{array}{lcl} \Phi(s_i, s_i) & = & [\![s_i]\!] \\ \Phi(s_i, n) & = & \sqcup_{(m,n)\in E}\ \mathcal{E}(n) \circ \Phi(s_i, m) \end{array}$$

and $\mathcal{E}$ is defined as either the normal transfer function for intraprocedural edge or the Φ for interprocedural ones:

$$\mathcal{E}(n) = \begin{cases} [\![n]\!] & if\ \forall j : n \neq s_j \\ \Phi(s_j, e_j) & if\ \exists j : n = s_j \end{cases}$$

The results of the above equation system are used to compute the solution of the data flow problem:

$$\begin{array}{lcl} \mathcal{F}(s_{main}) & = & [\![s_{main}]\!](\perp) \\ \mathcal{F}(s_i) & = & \sqcup_{(c,s_i)\in E}\mathcal{F}(c) \\ \mathcal{F}(n) & = & \Phi(s_j, n)(\mathcal{F}(s_j))\ for\ n \neq s_j, n \in V_j \end{array}$$

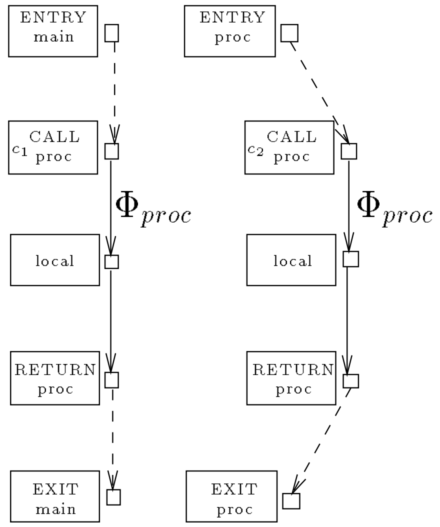

Figure 2: Functional Approach

2.3 Comparison

We briefly summarize the properties of both approaches (assuming a language with recursive procedures and table based implementation of the functional approach) for a given lattice $\mathcal{L}$. While the call string approach computes only values of $\mathcal{L}$, the functional approach computes values from $\mathcal{L} \rightarrow \mathcal{L}$. For guaranteeing termination it is required that this function space is finite. So the callstring approach is always guaranteed to terminate, but may deliver poorer results. Additionally one has the problem of finding an appropriate length K.
The functional approach is guaranteed to terminate for finite lattices, but may be exponential in the size of the lattice, and delivers best results.

3 Practical Evaluation

We generated data flow analyzers for *copy constant propagation*(*ccp*) which is a *distributive* problem and for *full constant propagation*(*fp*) which is not. $\mathcal{L}_{ccp}$ is finite ($\Rightarrow$ bounded height) and the $\mathcal{L}_{fp}$ is infinite but has bounded height. The *ccp* only processes assignments of the form $x := y$; or $x := c$ where x, y are variables and c is a constant, whereas *fp* includes a full interpreter for expressions. Furthermore, we use a simplified form of a technique called *downset completion* [5]; instead of combining the data flow information directly where two (or more) control flow edges come together, we first apply the transfer function and combine the results: $f(a \sqcup b)$ is replaced by $f(a) \sqcup f(b)$. This results in better solutions for non-exact $\sqcup$.

Table 1 shows the structure and size of the used test programs (a * indicates programs with procedural variables). The columns give from left to right: the name of the program, a short description, the number of lines without comments, the number of procedures, the number of control flow nodes,the number of its variables and the table where the measurements can be found.

We applied for each of the programs the *functional* analyzer. It finds the precise solution (MOP) for any program in case of the `copy constant propagation`. This is also true for the **`full constant propagation`** but the termination of the functional analyzer is not guaranteed. We applied also the *call string* analyzers up to the call string length, where they find the same set of available constants and compared their run times. For each program we have a table consisting of five columns: The first one contains the applied method, $\mathcal{F}$ for the functional analyzers and $\mathcal{C}_K$ for the *call string* analyzers with call string length K. As a measurement of precision the second column contains the number of available constants in the program. This is the number of constant variables at all control flow nodes, i.e.

$$\sum_{n \in V} |\{var | (\sqcup_{i=1,\ldots,Arity(n)} flow(n)[i])(var) \neq \bot, \top\}|$$

program	description	lines	flow nodes	procedures	variables	table
bison	parser generator	6438	11722	155	3575	9
cdecl	C++ compiler part	2831	2401	33	841	3
cloop	benchmark	1488	2176	26	622	8
dhry	dhrystone	446	319	14	240	6
ed	editor	1506	2745	47	796	5
flex	scanner generator	5985	8688	129	2314	7
flops	benchmark	723	353	3	139	2
gzip*	compress	4056	6043	47	796	12
linpack	benchmark	821	796	13	278	11
twig	code generator	2092	3198	81	1085	4
xmodem	communication	2060	2965	31	1159	10

Table 1: The set of test programs

The next column shows the number of interprocedural constants: variables that have a constant value at a procedure entry:

$$\sum_{s_p \in V_p} |\{var|(\sqcup_{i=1,\ldots,Arity(s_p)} flow(s_p)[i])(var) \neq \bot, \top\}|$$

The fourth column gives the number of source code transformations that can be done with the information obtained by the constant propagation. This is written in the form $x(y)$ with y being the number of replacements of variables by their values and x being the number of resulting (algebraic) transformations: e.g. if the expression $(x + 10 + z)$ is transformed to $(15 + z)$ by replacing x with 5 we would write that as 1(1), 1 for the replacement of x and 1 for the transformation of (5+10) to 15. The next column shows the number of steps in our fixpoint iterator and the last gives the runtime in seconds on a SUN SPARCsystem-600 (Ross-RT625 cpu,128Mb memory), running SunOS 4.1.4.

3.1 Results

As we can see, for some programs the functional analyzer is most suitable, whereas for other programs it doesn't find more constants than the $\mathcal{C}_0$ analyzer .
We have summarized our results in table 13 for the copy constant propagation and in table 14 for the full constant propagation. We have determined the best strategy finding all available constants (second column) and the best for source code folding (third column).

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	6897	7	107(101)	538	0,52
$\mathcal{C}_1$	7973	7	119(107)	692	0,50
$\mathcal{F}$	7973	7	119(107)	572	0,70
full constant propagation					
$\mathcal{C}_0$	7080	7	188(175)	582	0,74
$\mathcal{C}_1$	7306	7	217(184)	595	0,58
$\mathcal{F}$	7306	7	217(184)	515	0,67

$\mathcal{C}_1$ produces the same results as $\mathcal{F}$ in shorter time for both data flow problems and is better than $\mathcal{C}_0$ in precision *and* speed.

Table 2: `flops.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	615	0	10(3)	3807	15,66
$\mathcal{C}_1$	627	0	10(3)	10846	66,41
$\mathcal{F}$	627	0	10(3)	12144	113,93
full constant propagation					
$\mathcal{C}_0$	885	0	10(3)	3705	15,93
$\mathcal{C}_1$	903	0	10(3)	10612	65,08
$\mathcal{C}_2$	911	0	10(3)	23314	135,02
$\mathcal{F}$	911	0	10(3)	11688	101,50

The simple $\mathcal{C}_0$ analyzers find all relevant constants for source code optimization. However, $\mathcal{F}$ and $\mathcal{C}_1$ find more available constants; the former in shorter time. For *full constant propagation* $\mathcal{C}_2$ computes the same results as $\mathcal{F}$, but needs more time.

Table 3: `cdecl.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	756	0	6(3)	7450	6,83
$\mathcal{C}_1$	865	0	6(3)	43051	45,13
$\mathcal{C}_2$	891	0	6(3)	154242	308,22
$\mathcal{F}$	993	0	6(3)	327395	1746,14
full constant propagation					
$\mathcal{C}_0$	995	0	6(3)	7217	8,31
$\mathcal{C}_1$	1130	0	6(3)	42741	47,05
$\mathcal{C}_2$	1156	0	6(3)	153081	230,13
$\mathcal{F}$	1258	0	6(3)	322057	1813,02

$\mathcal{C}_0$ finds same foldings as all other tried call string analyzers and the functional ones. No interprocedural constants are found.

Table 4: `twig.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	1098	1	42(14)	6289	7,81
$\mathcal{C}_1$	1175	1	44(16)	13151	26,04
$\mathcal{C}_2$	1198	1	44(16)	21247	37,05
$\mathcal{C}_3$	1198	1	44(16)	32131	69,25
$\mathcal{F}$	1198	1	44(16)	17664	27,25
full constant propagation					
$\mathcal{C}_0$	2503	3	50(18)	6061	6,99
$\mathcal{C}_1$	2714	3	52(20)	12839	24,59
$\mathcal{C}_2$	2763	3	52(20)	19878	30,63
$\mathcal{C}_3$	2763	3	52(20)	30624	53,25
$\mathcal{F}$	2763	3	52(20)	16140	21,85

$\mathcal{C}_2$ has same precision as $\mathcal{F}$, but runs slower. For *copy cp*, $\mathcal{C}_1$ computes foldings faster than $\mathcal{F}$. $\mathcal{C}_3$ computes same information like $\mathcal{C}_2$ does, but runs slower.

Table 5: `ed.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	81	5	2(2)	525	1,82
$\mathcal{C}_1$	109	5	2(2)	613	1,54
$\mathcal{C}_2$	109	5	2(2)	617	1,45
$\mathcal{F}$	109	5	2(2)	710	2,06
full constant propagation					
$\mathcal{C}_0$	87	5	3(3)	481	1,26
$\mathcal{C}_1$	141	5	15(11)	567	1,49
$\mathcal{C}_2$	141	5	15(11)	571	1,43
$\mathcal{F}$	141	5	15(11)	631	1,49

$\mathcal{C}_2$ computes precise information in shortest time.

Table 6: `dhry.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	3362	41	22(60)	15404	16,40
$\mathcal{C}_1$	3496	42	22(60)	47366	88,76
$\mathcal{C}_2$	9714	116	22(83)	113941	378,66
$\mathcal{C}_4$	9724	116	22(83)	180127	1966,37
$\mathcal{F}$	9731	117	22(83)	75616	193,54
full constant propagation					
$\mathcal{C}_0$	4329	41	22(60)	15230	16,82
$\mathcal{C}_1$	4597	42	22(60)	47040	94,54
$\mathcal{C}_2$	10890	116	24(84)	47040	397,14
$\mathcal{C}_4$	10908	116	24(84)	183071	2024,36
$\mathcal{F}$	10911	117	24(84)	74486	209,26

$\mathcal{F}$ computes most precise results in shortest time. *full constant propagation* finds more foldings than *copy* cp.

Table 7: `flex.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	9839	127	145(56)	6360	20,95
$\mathcal{C}_1$	9865	127	145(56)	34420	200,30
$\mathcal{F}$	19649	149	145(56)	59940	6309,13
full constant propagation					
$\mathcal{C}_0$	10367	131	149(60)	6417	21,90
$\mathcal{C}_1$	10711	131	149(60)	44654	260,06
$\mathcal{C}_2$	10803	131	149(60)	164940	728,21
$\mathcal{F}$	21952	151	151(61)	48990	2402,53

The $\mathcal{C}_0$ analyzers find all foldings, whereas the number of available constants is much greater for more complex analyzers.

Table 8: `cloop.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	2699	11	10(5)	22918	24,45
$\mathcal{C}_1$	2974	11	10(5)	70052	157,12
$\mathcal{C}_2$	2978	11	10(5)	152398	372,43
$\mathcal{F}$	2978	11	10(5)	249544	303,74
full constant propagation					
$\mathcal{C}_0$	5591	11	10(5)	22193	24,03
$\mathcal{C}_1$	5960	11	10(5)	68347	152,13
$\mathcal{C}_2$	5973	11	10(5)	221592	1674,14
$\mathcal{F}$					∞

The $\mathcal{F}$ for *full cp* needed more than 8192 table entries for an abstract function. Whether we really found a practical example of non termination is currently under investigation. This program only contains *copy* constants.

Table 9: `bison.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	2716	36	6(6)	7822	88,77
$\mathcal{C}_1$	3269	36	6(6)	22023	256,61
$\mathcal{C}_2$	3269	36	6(6)	37709	326,55
$\mathcal{C}_4$	3269	36	6(6)	75498	427,72
$\mathcal{C}_5$	3269	36	6(6)	108974	526,94
$\mathcal{F}$	4806	36	6(6)	31402	802,53
full constant propagation					
$\mathcal{C}_0$	3870	36	12(12)	7769	94,23
$\mathcal{C}_1$	4756	36	12(12)	20770	226,18
$\mathcal{C}_2$	4756	36	12(12)	36299	323,80
$\mathcal{C}_4$	4756	36	12(12)	72720	369,64
$\mathcal{C}_5$	4756	36	12(12)	104181	483,13
$\mathcal{F}$	7521	36	12(12)	28393	593,11

fp finds twice as much foldings as *copy* cp. The number of foldings is equal for each analyzer, whereas the preciseness increases with the call string length. Unfortunately, the number of call strings of length greater than 6 results in large memory consumption and cannot be handled currently.

Table 10: `xmodem.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	355	6	78(18)	1483	5,17
$\mathcal{C}_1$	355	6	78(18)	4672	17,60
$\mathcal{C}_2$	355	6	78(18)	7383	24,36
$\mathcal{F}$	355	6	78(18)	1856	5,72
full constant propagation					
$\mathcal{C}_0$	385	6	80(21)	1322	4,57
$\mathcal{C}_1$	385	6	80(21)	4641	10,15
$\mathcal{C}_2$	385	6	80(21)	6012	16,64
$\mathcal{F}$	385	6	80(21)	1703	5,31

fp finds more available constants and foldings than the *ccp*. In all cases $\mathcal{C}_0$ is the most efficient technique.

Table 11: `linpack.c`

Method	Precision	Ipc	Foldings	Steps	Time
copy constant propagation					
$\mathcal{C}_0$	2744	0	16(34)	13213	9,52
$\mathcal{C}_1$	2822	0	16(34)	74812	91,68
$\mathcal{C}_2$	2822	0	16(34)	449596	1122,28
$\mathcal{F}$					∞
full constant propagation					
$\mathcal{C}_0$	4117	0	24(36)	11538	8,53
$\mathcal{C}_1$	4916	0	24(36)	71968	83,00
$\mathcal{C}_2$	5321	0	24(36)	489354	1604,51
$\mathcal{F}$					∞

Although the functional approach for the *copy* cp is guaranteed to terminate, both functional analyzers do not terminate within two hours and produce function tables with more than 8192 entries. Is is possibly caused by the procedure variables which make the possible flow of control very complex. $\mathcal{C}_0$ finds same number of foldings as all other call string analyzers we have used.

Table 12: `gzip.c`

Program	Available	Foldings
bison.c	$\mathcal{F}$	$\mathcal{C}_0$
cdecl.c	$\mathcal{C}_1$	$\mathcal{C}_0$
cloop.c	$\mathcal{F}$	$\mathcal{C}_0$
dhry.c	$\mathcal{C}_2$	$\mathcal{C}_2$
ed.c	$\mathcal{F}$	$\mathcal{C}_1$
flex.c	$\mathcal{F}$	$\mathcal{F}$
flops.c	$\mathcal{C}_1$	$\mathcal{C}_1$
gzip.c	$\mathcal{C}_1$	$\mathcal{C}_0$
linpack.c	$\mathcal{C}_0$	$\mathcal{C}_0$
xmodem.c	$\mathcal{F}$	$\mathcal{C}_0$

Evaluation: number of winnings		
Method	Available	Foldings
$\mathcal{F}$	5	1
$\mathcal{C}_0$	1	6
$\mathcal{C}_1$	3	2
$\mathcal{C}_2$	1	1

This table lists the winner of both available constants and foldings for each program. The functional approach finds in 50% the available constants in shortest time, whereas a (simple) $\mathcal{C}_0$ analyzer computes in 60% all foldings.

Table 13: `copy constant propagation`

Program	Available	Foldings
bison.c	$\mathcal{C}_2$	$\mathcal{C}_0$
cdecl.c	$\mathcal{F}$	$\mathcal{C}_0$
cloop.c	$\mathcal{F}$	$\mathcal{F}$
dhry.c	$\mathcal{C}_2$	$\mathcal{C}_2$
ed.c	$\mathcal{F}$	$\mathcal{F}$
flex.c	$\mathcal{F}$	$\mathcal{F}$
flops.c	$\mathcal{C}_1$	$\mathcal{C}_1$
gzip.c	$\mathcal{C}_2$	$\mathcal{C}_0$
linpack.c	$\mathcal{C}_0$	$\mathcal{C}_0$
xmodem.c	$\mathcal{F}$	$\mathcal{C}_0$

Evaluation: number of winnings		
Method	Available	Foldings
$\mathcal{F}$	5	3
$\mathcal{C}_0$	0	5
$\mathcal{C}_1$	2	1
$\mathcal{C}_2$	3	1

The functional approach finds in 50% the available constants in shortest time, whereas a (simple) $\mathcal{C}_0$ analyzer computes in 50% all foldings (about the same results as in table 13).

Table 14: `full constant propagation`

4 Conclusion and Further Work

Although no new ideas are presented in this paper, it presents a rather fair practical evaluation of the two techniques *call string*– and *functional approach* for interprocedural data flow analysis applied to two data flow problems. We have shown that the theoretic results (non termination) do occur in practise. We also have shown that a full constant propagation delivers better results than a simple one (Table 10). Furthermore one can see that the set of call strings which should be kept separately is small for real programs, but leads to similar results as the functional approach.

So we can draw the conclusion that an expensive functional approach can be replaced by a cheaper call string approach, at least for constant propagation in languages similar to C. The analyzer generator PAG can be used to determine automatically the relevant call string length for each analysis. A related call string analyzer can be incorporated into a compiler for this language.
In the future we will test if these results also hold for other analyses too.

References

[1] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In Radhia Cousot and David A. Schmidt, editors, *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer, September 1996.

[2] Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In Alan Mycroft, editor, *SAS'95, Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 33–50. Springer, September 1995.

[3] 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.*

[4] Anita L. Chow and Andres Rudmik. The design of a data flow analyzer. *SIGPLAN Notices*, 17(6):106–113, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

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

[6] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. *SIGPLAN Notices*, 28(6):90–99, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[7] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407, San Francisco, California, January 1995.

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

[9] 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.

[10] 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.

[11] Robert Metzger and Sean Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 2(1–4):213–232, March–December 1993.

[12] 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.

[13] Arnd Poetzsch-Heffter. Interprocedural data flow analysis based on temporal specifications. Technical Report CORNELLCS//TR93-1397, Cornell University, Computer Science Department, October 1993.

[14] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, January 1995.

[15] 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.

[16] Stephan Thesing, Florian Martin, and Martin Alt. *PAG User's Manual*, 1997.

[17] Steven W. K. Tjiang and John L. Hennessy. Sharlit – a tool for building optimizers. *SIGPLAN Notices*, 27(7):82–93, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

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