Data Flow Analysis applied to optimize generic workflow problems

Paulo Jorge Matos
Departamento de Informática e Comunicações
Escola Superior de Tecnologia e de Gestão
Instituto Politécnico de Bragança
Campus de Santa Apolónia
5300 Bragança, Portugal
Mail: pmatos@ipb.pt
Phone: +351 273303082
Fax: +351 273313051

Pedro Rangel Henriques
Departamento de Informática
Universidade do Minho
Gualtar
4710 Braga, Portugal
Mail: prh@di.uminho.pt
Phone: +351 253604470
Fax: +351 253604471

Abstract.
The compiler process, the one that transforms a program in a high level language into assembly or binary code, is a much elaborated process that mixes several powerful technologies, some of them developed specifically for this area. Nowadays, compilers are highly developed systems that can analyze and improve quite efficiently the source code, profiting from all the potential of the new processor architectures. This paper introduces a common type of analysis - the Data Flow Analysis – that is used to compute flow-sensitive information about programs, whose results are essential to produce many code optimizations. It is also argued that the problem of analyzing the data flow in software programs has many similarities with the problems found in industrial engineering; planning and management. As consequence, it is possible to apply analysis and optimization techniques used by compilers in these areas.

Keywords: Data Flow Analysis, flowgraphs, techniques for industrial engineering.

1 Introduction

The main goal of a compiler is to identify the operations described in some high-level programming language (the source or input, language) - like Fortran, Pascal, C/C++, ML, etc - and convert them into a list of assembly or binary instructions (the target, or output, code) executable by the chosen processor (Pentium, PowerPc, Sparc, ...). This translation must be done without losing the program semantics, which means that the output code must do exactly what is described by the source code.

The theoretical development in computer sciences, namely in the area of (formal) programming methods, has increased the distance between the source languages and the machine code languages.

The languages have evolved, and nowadays they are more abstract, syntactically more complex, semantically more powerful, supporting programming paradigms that are quite different from the one that is actually available at the processors level (corresponding to the imperative style of the assembly or machine languages). This evolution requires, from the compiler developers, more powerful solutions that, by one side, solve efficiently the new syntactic and semantic problems and, by the other side, obtain the total gain of the evolution occurred at the processors architecture, generating optimal code (as shorter and faster as possible).

The first tasks executed by a compiler are concerned with the analysis of the source language (lexical, syntactic and semantic), in order to recognize the meaning of the program; this phase inevitably produces an intermediate representation of the program that is more or less similar to the output code. At this intermediate level, the program is submitted to several optimizations to improve the output code quality. The subject of this paper is to show how some technical solutions used at this level of the compilation process, namely the Data Flow Analysis – DFA – (Kam and Ullman, 1976; Muchnick, 1997; Nielson *et al.*, 1999; Hecht, 1977) and the respective code optimizations, can be adapted to solve problems of other areas. This type of analysis does not produce any kind of transformation over the source program, but is responsible for collecting information, essential to some optimization strategies.

Figure 1 shows the use of a compiler, emphasizing its internal phases. The *Program Developer* is the person that writes the *Source Program* and submits it to the *Compiler* to obtain the *Output Program* (executable file). So, the function of the compiler is to recognize the source program (a sequence of characters that obey to the grammar rules of the source language), and then generate the output code. The interpretation is done by the lexical, syntactic and semantic analyzers, which form the *front-end* (FE) of the compiler (the FE groups the components that are dependent of the source language); the code generation consists essentially of two processes - the register allocation and the instruction selection - that constitutes the *back-end* (BE) of the compiler (the BE gathers the components that are dependent of the computer architecture).

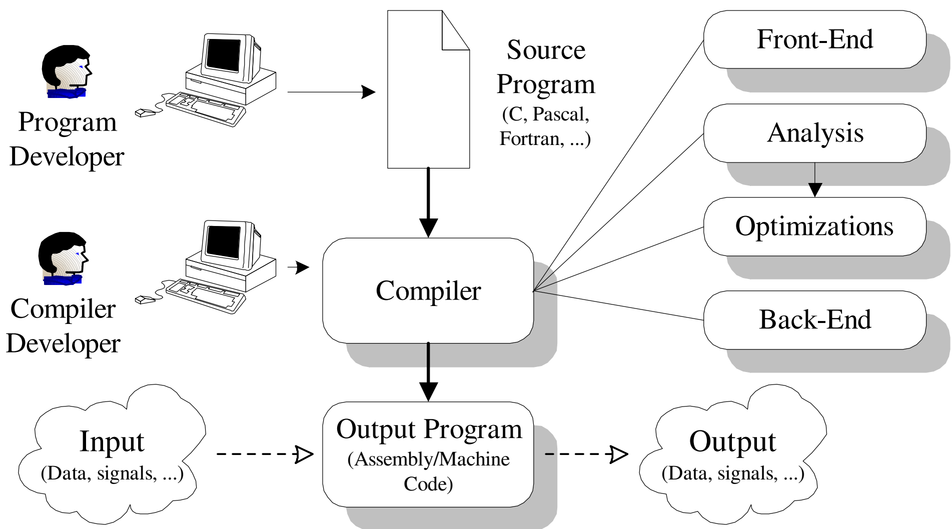


Figure 1 - Compiling and developing processes.

It is also normal to have a *middle-part* that contains all components that are independent of the characteristics of the source language and the computer architecture (which turn easier their reuse

when developing new compilers). The components commonly implemented at the *middle-part* are the several types of flow analysis (data or code), and many code optimization routines.

In this paper, we are mainly concerned with Data Flow Analysis, whose goal is to obtain information about the use of variables and values, that are sensitive to the execution flow of the program. To implement this analysis, the information is typically represented as a flow-graph, where the edges correspond to the execution flow of the program, and the vertices to the operations that directly or indirectly manipulate variables and values.

Notice that any imperative source language contains the necessary elements to build easily this type of representation:

- The flow-graph is directly extracted from the control structures of the language (sequential, conditional and loop statements);
- The operations that produced and manipulated data are well-defined (load/store, read/write, arithmetic, logical, …).

But the DFA is not restricted to imperative languages, since the compilation process can produce the same imperative intermediate representation even from languages of other programming paradigms.

The next section shows an example of code optimization that requires the execution of data flow analysis. It is demonstrated the benefits of this kind of optimization; explained how this analysis can be implemented and how the solution can be generalized for any type of information. In section three, it is demonstrated that the elements used for code representation, specially at the intermediate and low levels, have some similarities with the ones that are required for the representation of engineering and management problems. It is also explained how the approach followed in compilation can be applied to solve these problems. The last section presents the conclusions and describes the future work.

It is important to notice that the examples used along the paper are purposely small, very simple and do not require special knowledge about compilation or even about programming languages. The objective is to facilitate the understanding of the subject; however this does not mean that the real examples are so easy like these or that the code optimizations are limited to these transformations.

2 Data Flow Analysis

As was already told, Data Flow Analysis is responsible for collecting information about definition and use of variables and values (data handling) along a given source program; that information is then used by the optimization routines to minimize the code length or make its execution faster. To explain how this is done and why this type of analysis is necessary, we will present one example where the *Common Sub-Expressions Elimination* technique is applied. This optimization identifies and removes expressions that are duplicated in the source program.

The first example, shown in Figure 2(a), is a short fragment of a C program where the same sub-expression, $i+1$, is computed several times (op_2, op_3, op_4, op_5, op_7,op_8).

Unnecessary computations can be avoided if every time that the same sub-expression appears, computing exactly the same value, is replaced by a variable holding that value obtained the first time that the sub-expression was evaluated.

Applying the *Common Sub-Expressions Elimination* strategy to the program of Figure 2(a), we obtain the program of Figure 2(b). Notice that the occurrence op_5 of the sub-expression $i+1$ is not replaced, because at this position the value of the variable i was already changed (by the operation op_4). The same happens with its occurrence at op_8, but now because the sub-expression may have different values depending on the actual execution path (it has a direct influence in the value of variable i when it reaches operation op_8). As the compiler can not change the semantics of the source program, a sub-expression should only be replaced when it is possible to determine that such operation is safe.

```
op1    …
op2    a = i + 1;
op3    if(i+1>0){
op4      i = i + 1;
op5      b = i +1;
op6    }else
op7      b = 2*(i+1);
op8    c = (i+1)/2;
op9    …
```

(a) Code before the optimization.

```
op1    …
op2    a = i + 1;
op3    if(a>0){
op4      i = a;
op5      b = i + 1
op6    } else
op7      b = 2 * a;
op8    c = (i+1)/2;
op9    …
```

(b) Code after the optimization

Figure 2 - The Common Sub-Expression Elimination optimization.

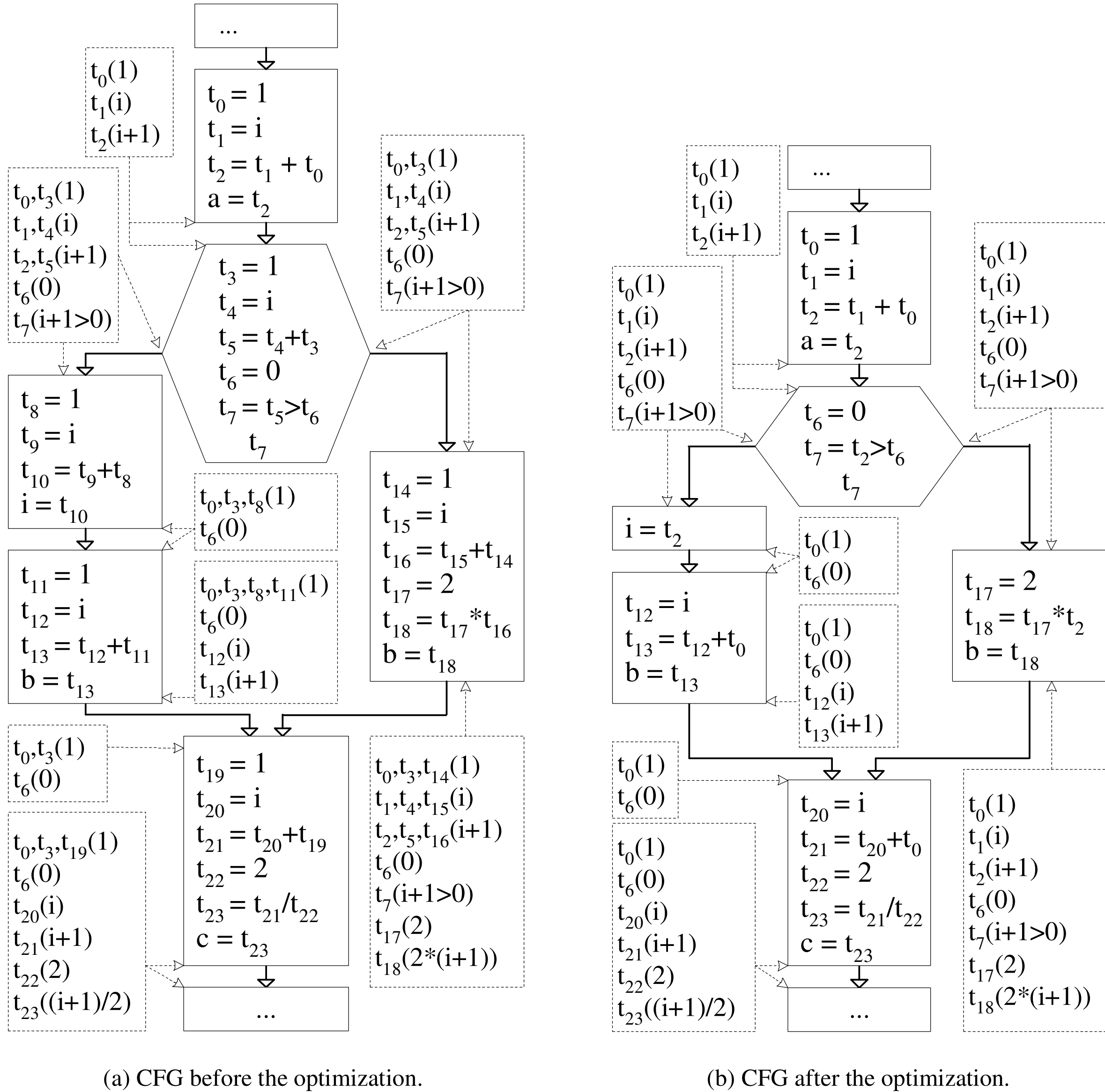
The optimization illustrated in the example above (Figure 2) dependents on the control of the program flow, which means that the optimization routine must consider the several paths that may occur during the execution of the program. The DFA comes to scene just to solve this problem, feeding the system with information concerned with the computation and propagation of values. Moreover, the DFA (that is the generic name of this type of analysis) is intended to work over the intermediate representation in order to keep the algorithms independent of the source language. In the present case, *Common Sub-Expression Elimination* technique, the *Reaching Definition Analysis* is used to know which sub-expressions are available at each position of the program.

Figure 3(a) shows a graphical representation, the Control Flow Graph (CFG), of the program of Figure 2(a), with more detailed information. Each expression inside a program block is decomposed, independently from the others, into elementary operations (including operands), and the result of each one is hold by a temporary variable (t_i). The list with all the available sub-expressions, so far obtained, is associated with the start and the end of each block (vertex of the graph). Each element of this list contains a list of temporary variables (one or more) and, between parentheses, the respective sub-expression, according to the following rules:

List_element → *List_Temp_Variables '(' Sub-Expression ')'*
List_Temp_Variables → *Temp_Variable* | *List_Temp_Variables ',' Temp_Variable*

It is assumed that the start list of the initial block is empty.

Figure 3(b) shows the effects of the *Common Sub-Expression Elimination* strategy using the information provided by the analysis. For the moment, the important is to observe that in fact this optimization (as consequence of the DFA performed) improved significantly the quality of the output code.



(a) CFG before the optimization. (b) CFG after the optimization.

Figure 3 - A CFG with the result of Reaching Definition Analysis.

Now we describe in detail how the analysis is done, considering just one node n of the graph. Suppose that the list of available sub-expressions at the entry of n is $L_{entry}(n)$, and at the exit is $L_{exit}(n)$. The goal is to compute $L_{exit}(n)$ based on the values of $L_{entry}(n)$ and on the own contributions of node n. Initially $L_{exit}(n)$ takes the value of the $L_{entry}(n)$. Then it is necessary to analyze, one by one, the sub-expressions of n (starting on the first one), identifying the shape of the sub-expression. If it has the form t_i = *expr,* it is necessary to test if *expr* already exists in $L_{exit}(n)$; if true, then t_i is appended to the list of temporary variables associated with *expr*; if false, then a new element, of the form $t_i(expr)$, is

append to $L_{exit}(n)$. If the sub-expressions has the form $var = t_j$, it is necessary to remove all elements whose sub-expression contains one or more references to *var*.

Notice that when a variable is defined (a value is assigned to it), its previous value will probably change; as consequence, all sub-expressions using that variable, that were available at this point of the program, are no longer valid.

The description above can be formulated using equation 1, where the f_{expr} represents the contributions of the sub-expression *expr*.

$$L_{exit}(\text{expr}) = f_{\exp r}(L_{entry}(\text{expr})), \qquad \text{where } expr \in n \tag{1}$$

If we associate the $L_{entry}(n)$ with the L_{entry} of the initial sub-expression of n, and $L_{exit}(n)$ with the L_{exit} of the last sub-expression of n, then it is possible to compute directly $L_{exit}(n)$ based on the $L_{entry}(n)$, using the equation 2, where f_n is the composition of the $f_{expr}()$ of all sub-expressions of n, starting on the last one.

$$L_{exit}(n) = f_n(L_{entry}(n)), \qquad f_n = f_{\exp r_{last}} \circ \ldots \circ f_{\exp r_{first}} \tag{2}$$

Now let us see how the analysis is done for the full graph. If n has only one predecessor, m, then $L_{entry}(n)$ takes the value of $L_{exit}(m)$. If n has more than one predecessor, then it is necessary to join the several L_{exit} of the predecessors of n to compute $L_{entry}(n)$. This corresponds to assign to $L_{entry}(n)$ the elements that are common to all L_{exit} of the predecessor nodes. It is possible to formulate this using equation 3, where **Flow** represents the edges between nodes.

It is important to notice that the edges that belong to **Flow** have not to be the same ones that are used at the flow-graph; it depends on the direction of the analysis that can be *forward*, if is done following the same direction of the edges of the flow-graph, or *backward* if is done in the reverse direction. The symbol $\coprod$ represents the *join* operation that defines how to combine the lists *L()* that reach a node.

$$L_{entry}(n) = \coprod \{L_{exit}(n') \mid (n', n) \in Flow\} \tag{3}$$

Equation 3 can be redefined into equation 4 in such way that it is possible to have a non-empty list, l_{initial},, at the entry of the initial block of the graph.

$$L_{entry}(n) = \begin{cases} l_{initial} & n \in \textit{Initial Nodes} \\ \coprod \{\mathrm{L}_{\mathrm{exit}}(n') \mid (n', n) \in Flow\} & \text{Otherwise} \end{cases} \tag{4}$$

Now that we know how to compute the values for the nodes of the graph, it is still necessary to determine the order by which they should be processed. The solution that is proposed here is not the most efficient but the most versatile, which is essential for the subject of this paper: we suggest the use of an Iterative Data Flow Algorithm (IDFA), shown in Figure 4, that applies the equations, over and over, until no more changes occur at the lists associated with the nodes of the graph. The algorithm receives as input just one parameter, the flow-graph *g,* and returns as output a dictionary mapping nodes to lists of sub-expressions available at the entry of the nodes (the lists of sub-expressions available at the exit of the nodes can be easily computed using equation 2).

It is quite simple to generalize this DFA solution to solve similar problems (Dwyer, 1995; Knoop, Ruthing and Steffen, 1994). The algorithmic solution is the same one and the equations 1, 2, 3 and 4 are still valid. Essentially, it is only necessary to redefine the type of data structure that should be associated with the nodes, the functions associated with the sub-expression ($f_{expr}(\ )$), and the join operator.

```
Procedure IDFA( g : FlowGraph) : Dictionary(Node,List)
n, n' : Nodes
joinresult : List
Worklist : Set(Node)
L_entry, l_exit : Dictionary(Node,List)
Begin
        L_entry(g.InitialBlock ) = l_initial
        Worklist := g.Nodes - g.InitialBlock
        For each n ∈ g.Nodes do
                L_entry(n) := <>
        While Worklist ≠ ∅
                n := first(Worklist)
                Worklist -= {n}
                joinresult := <>
                For each n' ∈ Predecessor(n) do
                        l_exit(n') := f_n'(L_entry(n'))
                        joinresult ∪ = l_exit(n')
                If L_entry(n) ≠ joinresult then
                        L_entry(n) := joinresult
                        Worklist ∪ = {n}
        return L_entry
End
```

Figure 4 - The Iterative DFA Algorithm.

3 Applying DFA to another problems

This section is intended to demonstrate our proposal of extending the Data Flow Analysis to solve problems in areas not directly related to compilation. The first part of the section characterizes the set of processes to which we can apply the approach; the second part discusses two possible applications.

The processes, for which we forecast a possible application of the DFA, can be characterized as follows:

- They are quite big and hardly analyzed by hand;
- They can be modeled by a finite flow-graph, where the vertices represent the entities that operate the information under analysis, and the edges describe the information flow between operations;
- The information that is analyzed by the DFA should be flow-sensitive;
- It should be possible to define an algebraic data structure to represent the information, which form a complete lattice (see Appendix A);

- It should exist a finite set of functions that describe how the entities affect the information. These functions are designated by "transfer functions" and should be monotone (see Appendix A);
- It should be possible to define the identity function (the one that has not any effects over the information).

Figure 5 shows a flow-graph describing a generic industrial process complying with the characteristics above. The two DFA application examples that will be discussed in the rest of the section are both based on that model.

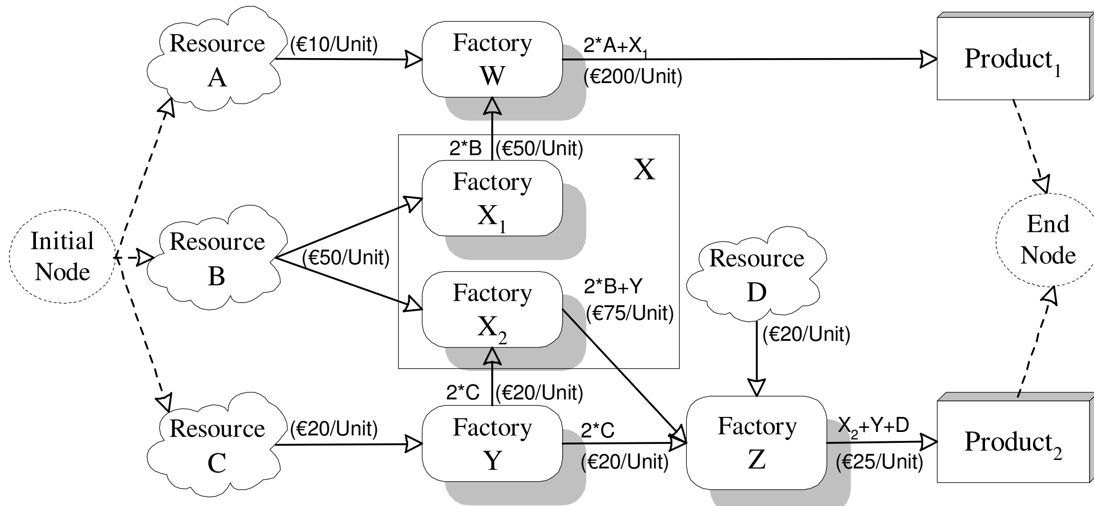


Figure 5 - A generic industrial process.

The original node *Factory X* was split into two new nodes *Factory* X_1 and *Factory* X_2, because it had two different outputs and in our model each node should have always the same exit value. To satisfy the formal aspects of the DFA technique, it is also necessary to introduce two extra nodes: the *Initial Node* and the *End Node* (see Appendix A).

3.1 Example 1

In the first case, we want to determine the resources that are present at each point of the process. This can be done using a set associated with the entry and the exit of each node, respectively Res_{entry} and Res_{exit}. The entry value for the *Initial Node* is the empty set, and the exit value for the *End Node* is the set with all resources available (it is expected that all resources are used by this industrial transformation). The analysis will follow the same direction of the flow of the process (*forward* direction).

As stated by equation 1, $Res_{exit}(n)$ is obtained applying the transfer function of n to the $Res_{entry}(n)$. Equation 5, below, defines the transfer function; in that case, its argument is the node itself, and not the set of the available resources at the entry of the node. So, equation 1 can be rewritten as the equation 6 bellow.

Res_{entry} is obtained applying the *join* operator to the sets of resources available at the exit point of the incoming edges. The *join* operation is the *union* of the resources provided by the predecessors of the node, as shown in equation 7.

$$f_{transfer}(n)=\begin{cases}\varnothing & n\in \text{Initial Nodes}\\ A & n=A\\ B & n=B\\ C & n=C\\ D & n=D\\ Res_{entry}(n) & \text{otherwise}\end{cases} \tag{5}$$

$$Res_{exit}(n)=f_{transfer}(n) \tag{6}$$

$$\mathrm{Res}_{entry}(n)=\begin{cases}\varnothing & n\in \text{Initial Nodes}\\ \coprod\{\mathrm{Res}_{exit}(n')\mid(n',n)\in Flow\} & \text{otherwise}\end{cases} \tag{7}$$

3.2 Example 2

The goal of the second case is to determine, at each point of the process, the cost of one production unit. One solution is to solve the problem by a conventional equation system, but it is also possible to use the DFA solution, even being a little more complex. Notice that in the DFA formulation, it is needed to represent the information at the entry of the node using just one data structure. This corresponds to have available the cost of each production unit at the entry of the node and not at correspondent input edges. So, it is necessary a data structure to hold information like: at the entry of node X_2 we have *B* at 50\$/unit, and the output product of node *Y* at $cost_y$/unit. This can be done by using a dictionary mapping products to cost/unit[1], which will be represented as $Costs_{entry}$ and is computed applying an equation similar to 7, where the *join* operator is the *union* of the dictionaries of the predecessor nodes.

$$f_{transfer}(n)=\begin{cases}\varnothing & n\in \text{Initial Nodes}\\ (A,10) & n=A\\ (B,50) & n=B\\ (C,20) & n=C\\ (D,20) & n=D\\ (W,2*A+X_1+200) & n=W\\ (X_1,2*B+50) & n=X_1\\ (X_2,2*B+Y+75) & n=X_2\\ (Y,2*C+20) & n=Y\\ (Z,X_2+Y+D+25) & n=Z\\ (Product_1,Z) & n=Product_1\\ (Product_2,W) & n=Product_2\\ (End\ Node,Product_1+Product_2) & n=End\ Node\end{cases} \tag{8}$$

[1] Since each node as only one output product, this one keeps the designation of the node.

At the exit of each node exists only one product (except for the *Initial* and *End* nodes). As consequence, the dictionary ($Cost_{exit}$) has only one element; for instance, the exit dictionary associated with node X_2 contains only the value (X_2, $Cost_{X_2}$). The value of $Cost_{X_2}$ is the result of applying the equation that quantifies the resources need to produce one unit of X_2 ($2*B+Y$), plus the cost of production of node X_2 (75/Unit). This operation is defined by the transfer function of the equation 8.

4 Conclusion

This is the first of a set of papers that we intend to publish about the adaptation of the *code analysis and optimization techniques*, implemented by most of the compilers, to solve different planning and management problems appearing in industrial, economical, governmental and information systems.

As explained along the paper, DFA is intensively applied at the development of compilers to support the code optimizations techniques. We believe that this kind of analysis can be used to solve many other problems with good results, and also that these problems probably are solved using other solutions that might be interesting for compiler developers.

At this point, some people may ask: why another solution that is much more complex than those already available and maybe not so efficient? We believe this solution has some advantages:

- The DFA solution is quite uniform and is easily adaptable;
- This approach can deal quite well with abstract information (not only with quantifiers);
- The solution works independently of the flow-graph topology, which means that it is not necessary a specific DFA for each problem, but only one for each class of problems (that determines the same type of information, using the same transfer function and the same join operator);
- Using the *structural analysis resolution*, instead of the IDFA algorithm of Figure 4, it is possible to modify the flow-graph and obtain the DFA results without have to compute all values again.

The focus of this paper is the Data Flow Analysis, but we have also in mind the idea of explore other compiler techniques - more elaborated forms of DFA (like the Alias Analysis), the Dependency Analysis (used for instruction scheduling), and Control Flow Analysis. We also expect to find out a better set of case studies (we are looking for new ones) to demonstrate the potential of this technology. Then, the next step will be the development of a tool/framework (Tjiang, 1993) to help the construction of the analysis and optimizations routines. This will be done based on a description of some characteristics of the problem - like the specification of the *join* operation or the definition of the *transfer* functions - and not on the description of the concrete problems.

References

Dwyer, M. (1995). Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs. PhD Thesis, Amherst, MA, USA.

Hecht, M. S. (1977). Flow analysis of computer programs. Elsevier North-Holland.

Kam, J. B. and Ullman, J. D. (1976). Global data flow analysis and iterative algorithms. Journal of the ACM, vol. 21, nº 3, pp. 158-171.

Knoop, J., Ruthing, O. and Steffen, B. (1994). Optimal Code Motion: Theory and Practice. ACM Transactions on Programming Languages and Systems, Vol. 16, 4, pp. 1117-1155.

Muchnick, S. (1997). Advanced Compiler Design and Implementation. Chapter 8, Morgan Kaufmann Publishers; ISBN: 1558603204.

Nielson, F., Nielson, H. and Hankin, (1999). Principles of Program Analysis. Chapter 2, Springer Verlag; ISBN: 3540654100.

Proctor, T. (1999). Creative Problem Solving for Managers, Routledge, London.

Tarjan , R. E. (1981). Fast algorithms for solving path problems. Journal of the Association for Computing Machinery, Vol. 28, n. 3, pp. 594-614.

Tjiang, S. (1993). Automatic Generation of Data-flow Analyzers: A tool for building optimizers. PhD Thesis, Stanford University, Computer Systems, Laboratory.

Appendix A

The DFA uses an algebraic structure, the complete lattice, to represent abstract proprieties of the variables, expressions or any other program constructor. The complete lattice is a special case of a partially ordered set, represented as $(L, \sqsubseteq)$, where:

- L is the set of values;
- $\sqsubseteq$ is the partial ordering relation[2] $\sqsubseteq : L \times L \rightarrow \{\text{true, false}\}$, that is:
  - reflexive (i.e. $\forall\, l : l \sqsubseteq l$),
  - transitive (i.e. $\forall\, l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$),
  - and anti-symmetric (i.e. $\forall\, l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$).

To explain what a complete lattice is, it is necessary to define some more concepts. So, let L be a partially ordered set and Y a non empty subset of L, then:

- $l \in L$ is a upper bound of Y if $\forall\, l' \in Y: l' \sqsubseteq l$;
- $l \in L$ is a lower bound of Y if $\forall\, l' \in Y: l \sqsubseteq l'$;
- $l \in L$ is Least Upper Bound (LUB) of Y, if l is a upper bound of Y and $l \sqsubseteq l_0$, whenever l_0 is any other upper bound of Y;
- $l \in L$ is Greatest Lower Bound (GLB) of Y, if l is a lower bound of Y and $l_0 \sqsubseteq l$, whenever l_0 is any other lower bound of Y.

It is not binding that Y has a LUB or a GLB, but when they exist they are unique and are denoted, respectively, as $\bigsqcup Y$ and $\bigsqcap Y$. The $\bigsqcup Y$ and the $\bigsqcap Y$ are, respectively, the join and meet operators, which satisfy the next proprieties:

- Closure : $\forall\, x, y \in L$, there is a unique z and $w \in L$, such that $x \sqcap y = z \wedge x \sqcup y = w$;
- Commutativity : $\forall\, x, y \in L, x \sqcap y = y \sqcap x \wedge x \sqcup y = y \sqcup x$;
- Associativity : $\forall\, x, y, z \in L, (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z) \wedge (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$.

A lattice is a partially ordered set $(L, \sqsubseteq)$ where the operators $\sqcup$ and $\sqcap$ are always defined. A lattice is fully represented by the tuple $L = (L, \sqsubseteq, \sqcup, \sqcap)$. A complete lattice is a lattice for which all non empty subset Y of L, have a LUB and GLB. As consequence a complete lattice has:

- A unique element $\bot \in L$, designate by bottom, such that $\bot = \bigsqcup \varnothing = \bigsqcap L$;
- And a unique element $\top \in L$, designate by top, such that $\top = \bigsqcap \varnothing = \bigsqcup L$.

The complete lattice can be represented by the tuple $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$.

---

[2] It is considered a partial ordering relation in opposition to a total ordering relation where $\forall\, x, y \in L, x \sqsubseteq y \vee y \sqsubseteq x$.

It is also important that the transfer functions used at the DFA satisfy the next conditions:

- All transfer functions should be monotone: $l \sqsubseteq l' \Rightarrow f_b(l) \sqsubseteq f_b(l')$. This means that an increase of the knowledge on l produced by $f_b(l)$ must give raise to an increase of knowledge on l' ($f_b(l')$);
- The set of all transfer functions, represented by $\mathcal{F}$, should contain the identity functions (for the blocks that have not effects over the lattice values);
- The set $\mathcal{F}$ should be closed under the composition of functions.