ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

M P A Sellink (Ed)

2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997

Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997

A Generic Architecture for Data Flow Analysis to Support Reverse Engineering

L. Moonen

Published in collaboration with the
British Computer Society

Springer

ISBN: 3-540-76228-0

A Generic Architecture for Data Flow Analysis to Support Reverse Engineering

Leon Moonen[*]

University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
`leon@wins.uva.nl`

Abstract

Data flow analysis is a process for collecting run-time information about data in programs without actually executing them. In this paper, we focus at the use of data flow analysis to support program understanding and reverse engineering. Data flow analysis is beneficial for these applications since the information obtained can be used to compute relationships between data objects in programs. These relations play a key role, for example, in the determination of the logical components of a system and their interaction.

The general support of program understanding and reverse engineering requires the ability to analyse a variety of source languages and the ability to combine the results of analysing multiple languages. We present a flexible and generic software architecture for describing and performing language-independent data flow analysis which allows such transparent multi-language analysis. All components of this architecture were formally specified.

Categories and Subject Description: D.2.2 [**Software Engineering**]: Tools and Techniques — Programmer workbench; D.2.6 [**Software Engineering**]: Programming Environments — Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance — Restructuring; D.2.m [**Software Engineering**]: Miscellaneous — Rapid prototyping; D.3.2 [**Software Engineering**]: Language Classifications — Data-flow languages

Additional Key Words and Phrases: Reverse engineering, Program understanding, Data flow analysis, Data flow representation language

1 Introduction

Data flow analysis [Hec77, Ken81] can be described as a process to collect information about the use, definition, and dependencies of data in programs. For example, consider the following program statement: $x := y + z$. Data flow analysis can be used to give answers to questions like: "What definitions of y might have determined the value of x used here?", "On which variables does this definition of x depend indirectly?", and "Where will the value of x computed here be used in the program?".

One can abstract from specific variables in these questions and ask this kind of information for all variables in a program. Such general questions are called *data flow analysis problems*. Their solution is described using *data flow analysis algorithms*. These algorithms capture program information and compute the derived (data flow) information.

Data flow analysis algorithms are frequently defined using operations on so-called *control flow graphs*. A control flow graph represents all possible execution paths of a given computer program: the nodes represent pieces of code and the edges represent possible control transfers between these code pieces. Labels are used to add extra program information to the nodes or edges of these graphs. Data flow analysis algorithms combine and manipulate this information using a set of equations that relate information at a given node to the information at other nodes. The solution of these equations yields the solution of a particular data flow analysis problem. A typical example of such a data flow equation

[*]The author was sponsored in part by bank ABN Amro, software house DPFinance, and the Dutch Ministry of Economical Affairs via the Senter Project #ITU95017 "SOS Resolver"

is $\mathsf{out}(n) = \mathsf{gen}(n) \cup (\mathsf{in}(n) - \mathsf{del}(n))$ which can be read as "the information at the output of node n is either generated in n or it enters at the input and is not deleted in n". In general, $\mathsf{in}(n)$ is a function that collects information from a subset of the neighbours of n, usually either the successors or the predecessors of n in the flow graph.

Data flow analysis is a *static* form of analysis: program information is collected without the actual execution of (parts of) the program. Therefore, it is not possible to determine the *exact* output of a program since it cannot be determined which execution path in the control flow graph is actually taken. Data flow analysis algorithms make approximations of this behaviour, for example, by considering both branches of an *if-then-else* statement and by performing a fixed-point computation for the body of a *while* statement. Such a fixed-point always exists because the data flow equations compute sets of variables and there are only a finite number of variables available since we only consider programs with a finite number statements. Therefore, there is a finite upper limit to the number of elements of the computed sets which means that a fixed-point always exists.

In terms of control flow graphs, static analysis means that *all* possible execution paths are considered to be actual execution paths. The result of this assumption is that one can only obtain approximate solutions for certain data flow analysis problems. Given a data flow analysis problem, such an approximate solution can err by *over-determination*: allowing conclusions which are stronger than actually valid, or it can err by *under-determination*: only allowing conclusions which are weaker than actually valid. Obviously, we prefer solutions that are valid if imprecise, so we will aim at the weaker conclusions. These are also called *safe* or *conservative* solutions.

Original applications of data flow analysis come from the area of compiler optimisation and include techniques such as support for dead code elimination, constant folding, common subexpression elimination, elimination of invariant code from loops and efficient register allocation [Hec77, ASU86]. Other applications include derivation of program information to support program development processes such as automated debugging [Agr91], testing [Har93] and maintenance [OCTS94, OCS94] of computer programs.

In this paper, we present generic architecture that supports language-independent specification and execution of data flow analysis algorithms. This architecture is based on two notions: First, the source language is separated from the *data flow representation language*: the language on which the data flow analysis is performed. This is similar to the use of an intermediate representation in the construction of compilers and the advantages are likewise: adding another language only requires adding another front-end which translates the languages into the intermediate representation. Second, we distinguish sub-levels of decreasing complexity in our data flow representation language and perform a number of transformations between these levels in order to simplify the language, and therefore simplify the data flow analysis. This facilitates the creation of standard, reusable, analysis components. Furthermore, it allows the definition of fresh data flow representation languages that are well suited for a given language and transform them into our basic data flow representation language.

2 Data Flow Analysis Preliminaries

In this section, we introduce the most commonly used notions in the field of data flow analysis. A more detailed description of these concepts can be found in [Hec77, Ken81, ASU86] which contain good introductions to data flow analysis and its application in compilers.

Data flow analysis attaches program information to control flow graphs using a labeling of the nodes and/or edges of the graph. Let $G = (N, E)$ be a graph and L a set of labels. A *node labeling* of G is a function $f : N \to L$ which maps each node to an element of L. Similarly an *edge labeling* of G is a function $g : E \to L$ which maps each edge to an element of L. In data flow analysis, these labels represent local information such as used or defined variables. Data flow analysis algorithms combine and manipulate this information by interpreting the labels as elements of an algebraic structure: the principal model for data flow analysis uses functions on a semi-lattice [MR90].

A *variable definition* is a reference to a variable that can modify the value of that variable. Thus, a variable is said to be defined if and only if it is referenced and its value can possibly be modified due to that reference. Examples of variable definitions are the occurrences of variables in input statements and in the left-hand sides of assignment statements. A *variable use* is a reference to a variable that cannot modify the value of that variable. Thus, a variable is said to be used if and only if it is referenced without possible modification (i.e., without being defined). Examples of variable uses are the occurrences of variables in output statements, conditions and in the right-hand sides of assignments.

For each node n,

$$\mathsf{rdtop}(n) = \begin{cases} \emptyset & \text{when } \mathsf{rdtop}(n) \\ \bigcup\limits_{m \in \mathsf{pred}(n)} \Big[\big[\mathsf{rdtop}(m) \cap \mathsf{pres}(m) \big] \cup \mathsf{def}(m) \Big] & \text{otherwise} \end{cases} \quad (1)$$

Figure 1: Data flow equations for definitions reaching the top

A path from node n to node m in a flow graph is called *definition-clear* with respect to a certain variable v if and only if there is no definition of v along that path. A definition d of a variable v in a node n is said to *reach* the bottom (respectively: top) of a given node m if and only if there is a definition-clear path from the definition in node n to the bottom (respectively: top) of node m. A definition d of a variable v is said to *kill* all definitions of variable v that reach d. A variable v is said to be *(a)live* at the bottom (respectively: top) of a node n if and only if there is a definition-clear path from the bottom (respectively: top) of n to a use of v in some node m.

The nodes in a control flow graph may represent pieces of program code that consist of multiple statements. Therefore, these nodes may contain multiple definitions of a given variable. However, since the granularity of a data flow analysis algorithm is equal to that of the nodes, we need to abstract from these definitions and consider only the variable definition that really matters for that node: the *locally exposed definition* (or *locally generated definition*) of a variable in a node is the last definition of that variable in that node. For each node n, let def(n) be the set of locally exposed definitions of variables defined in node n. Similarly, a *locally preserved definition* of a variable in a node n is a definition of a variable that is not killed in n. For each node n, let pres(n) be the set of locally preserved definitions of node n.

A *locally exposed use* of a variable v is a use of variable v in a node n which is not preceded by a definition of v in n. For each node n, let use(n) be the set of variables with locally exposed uses in node n. A *locally preserved variable* in a node n is a variable that is not killed in n. For each node n, let nodef(n) be the set of locally preserved variables of node n. Note that nodef(n) are exactly those variables that have a locally preserved definition (i.e., are defined in pres(n)).

3 Typical Data Flow Analysis Problems

In this section, we will describe some classical data flow analysis problems and present the data flow equations that can be used to solve them. Such equations are computed for every node in the control flow graph and if the graph contains cycles, a fixed-point computation is used. Since the problems presented here are intended as examples of data flow analysis problems, we assume that all necessary local data flow information is available in the program's control flow graph. For each data flow analysis problem that is described, we will present a brief example of its application. More information on these applications can be found in [ASU86].

3.1 Reaching Definitions

The *reaching definitions* at the top of a node n are the variable definitions that can reach the top of n. That is, the set of definitions that are preserved along the path from their definition to the top of n. This information can, for example, be used for the support of constant folding and dead code elimination. *Constant folding* is a compile time process which consists of deducing whether the value of an expression is a constant and if so, using that constant instead of the expression for code generation. Dead code elimination is a related notion. Program code is called *dead code* if it can never be reached during execution or when it computes values which are never used. An example of dead code is the code in the *then* branch of an *if-then-else* statement which has a condition that is always false. Both the *then* branch and the *if-then* statement can be eliminated, leaving only the code in the *else* branch.

The definitions reaching the top of a node n are all definitions that reach the top of predecessors of n and are preserved in n or the definitions that are locally generated in n. They can be computed using the equation in Figure 1.

3.2 Live Variables

The *live variables* at the bottom of a node *n* are the sets of variables that have a definition-clear path from the bottom of *n* to a use of *v* in some node *m*. Live variables can, for example, be used to determine optimal register allocation and assignment. *Register allocation* determines what values in a program should reside in registers and *register assignment* determines in which registers those values should reside.

The variables that are alive at the bottom of a node *n* are all variables that either are used or are alive at the bottom and not defined in successors of *n*. They can be computed using the equation in Figure 2. Note that this equation is a *bottom-up* equation whereas the equation for the reaching definitions problem was so-called *top-down* equation.

It is also possible to compute the definitions reaching the bottom of a node *n* and variables that are alive at the top of a node *n*. We refer to [Hec77] for more details.

3.3 Definition-Use Chaining

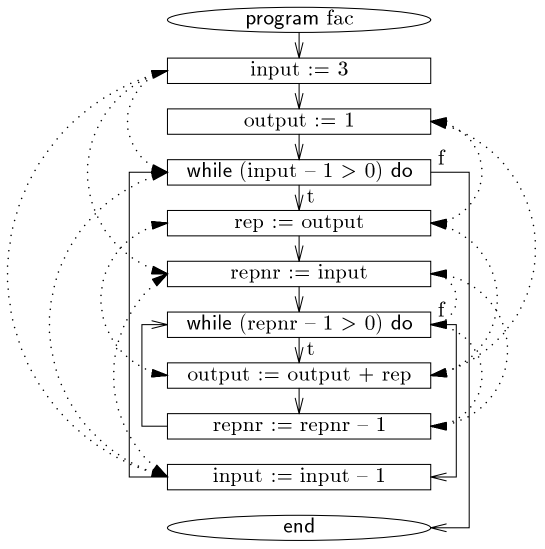
A data flow analysis problem which is very useful for the support of program understanding and reverse engineering is known as *definition-use chaining*. It allows us to relate each use of a variable to its possible definitions and each definition of a variable to its possible uses. This information can, for example, be used to track down year-2000 dependencies. Suppose we have the following information for each node $n \in N$:

– use(n)	the locally used variables of *n*;
– def(n)	the locally generated definitions of *n*;
– rdtop(n)	the definitions reaching the top of *n*;
– lvbot(n)	the variables that are alive at the bottom of *n*.

By considering both rdtop(n) and use(n) we can relate each use of a variable in a node to zero or more definitions reaching that node's top. Similarly, considering both lvbot(n) and def(n) we can relate each variable that is alive at the bottom of a node to zero or more definitions in that node. This double linking is called definition-use chaining. Using this information, we can relate each use of a variable to its possible definitions and each definition of a variable to its possible uses.

Example 3.1 In this example we present a small program in pseudo code and its control flow graph, annotated with the def-use chains for all variables.The program computes the factorial of the value in variable input and puts it in variable output. The figure on the right presents the control flow graph of this program (solid arrows), annotated with def-use chaining for all variables (dotted arrows). The characters 't' and 'f' denote the path that is taken when the condition of the while is respectively true or false.

```
program fac
begin
    declare input: natural,
            output: natural,
            repnr: natural,
            rep: natural;
    input := 3;
    output := 1;
    while (input – 1 > 0) do
        rep := output;
        repnr := input;
        while (repnr – 1 > 0) do
            output := output + rep;
            repnr := repnr – 1
        od;
        input := input – 1
    od
end
```



□

For each node n,

$$\mathsf{lvbot}(n) = \begin{cases} \emptyset & \text{when } \mathsf{lvbot}(n) \\ \bigcup\limits_{m \in \mathsf{succ}(n)} \Big[[\mathsf{lvbot}(m) \cap \mathsf{nodef}(m)] \cup \mathsf{use}(m) \Big] & \text{otherwise} \end{cases} \tag{2}$$

Figure 2: Data flow equations for live variables at the bottom

4 A Generic Architecture for Data Flow Analysis

Traditionally, data flow analysis algorithms are coded by hand and directed towards analysis of the specific programming language at hand. Obviously, this practice has some undesirable consequences:

- *Change of the language to be analysed results in re-implementation of all data flow analysis algorithms*. We expect this problem to grow in the near future due to the increased use of special purpose or proprietary languages with relatively short life cycles.
- *It is often difficult to combine the results of data flow analysis of different languages.* Transparent combination of the analysis results from different programming languages is, for example, very desirable for the analysis of legacy systems that are often written in multiple languages, e.g., COBOL and JCL.
- *Duplication of implementation efforts for standard components.* For example, a lot of data flow analysis algorithms are based on *interval analysis*, so the implementation of such a component is duplicated in a lot of analysers. Interval analysis is a technique based on control flow analysis to obtain the advantages of a syntax-directed analysis for general control flow graphs, even those that can not be analysed in a syntax-directed manner immediately [ASU86].
- *Information exchange between researchers and/or software practitioners is hindered by language specific approaches.*

To avoid these problems, we have developed a generic architecture that supports language-independent specification and execution of data flow analysis algorithms. The architecture consists of two major steps. First, we separate the source language from the *data flow representation language*: the language on which the data flow analysis is performed. This step is similar to the use of an intermediate representation in the construction of compilers and the advantages are likewise: adding another language only requires adding another front-end which translates the languages into the intermediate representation. Second, we distinguish sub-levels of decreasing complexity in our data flow representation language and perform a number of transformations between these levels in order to simplify the language, and therefore simplify the data flow analysis. This step facilitates the creation of standard, reusable, analysis components. Furthermore, it allows the definition of fresh data flow representation languages that are well suited for a given language and transform them into our basic data flow representation language.

4.1 Separate Abstraction and Analysis

The program information used by the data flow analysis algorithms can be seen as an abstraction of that program. For example, a typical data flow analysis problem like *def-use chaining* only uses information on which variables are defined and which variables are used by each statement. Other information, such as the operators that are used, is ignored. This abstraction property is reflected by the use of abstract interpretation for the implementation of data flow analysis [CR81, CC92, JN94]. *Abstract interpretation* is a technique for analysing programs in a given language by evaluating them with a non-standard semantics for that language [CC77]. However, since abstract interpretation is a language specific approach, the same software can not be reused for the analysis of other languages.

We propose to separate the abstraction and the data flow analysis phases by introducing a generic intermediate data flow representation level. The execution of the complete analysis process then consists of two consecutive steps (see Figure 3):

1. Abstract from programs in a given source language to programs in a data flow representation language. These programs contain all necessary data flow information.

2. Perform data flow analysis at the level of the data flow representation language.

The advantage of this approach is that all language dependent information is assembled in the component that performs the abstraction and that the data flow analysis can be completely language independent. As a result, all work that is done on the data flow analysis part can be re-used for a new source language whenever an abstraction for that language becomes available.

The disadvantage of this approach is that some program specific information is lost during abstraction. As a result this approach may be unsuitable for traditional data flow analysis applications such as compiler optimisation. However, this imposes no problems in our case since we use data flow analysis to support program understanding and reverse engineering which does not need that much program specific information.

The results of the data flow analysis are now expressed in terms of the data flow representation language. They can be linked back back to the original source code using *origin tracking*: an annotation technique which defines the relation between code in a translated (i.e., abstracted) program and its counterpart in the original program [DKT93]. Note that the use of origin tracking requires no additional implementation efforts when the abstraction is specified using ASF+SDF since it is currently being integrated with the ASF+SDF Meta-environment [Kli93].

4.2 Transformations on the Data Flow Representation Language

During the abstraction phase, we ignore information that is irrelevant for data flow analysis. Observe that this does not mean that the resulting data flow representation language is immediately suitable for data flow analysis. For example, the presence of unstructured flow of control in a program makes data flow analysis considerably more complex [Amm92]. However, abstraction from a language containing unstructured flow of control will not solve this problem since it results in a data flow representation language which still contains unstructured flow of control.

In order to reduce the complexity of program analysis such as data flow analysis, we need to simplify the syntax and structure of the data flow representation language. To facilitate such simplifications, we extend our architecture with a number of transformations that reduce the syntactical or structural complexity of the data flow representation language. As a result of this extension we no longer have a single data flow representation language, but instead we have a family of data flow representation languages with decreasing syntactical or structural complexity. Figure 4 gives an overview of the new architecture. In this picture the data flow representation languages are denoted by DFRL_i.

Based on some small case studies and analysis of existing data flow analysis algorithms, we have identified a number of transformations that frequently occur in data flow analysis software. Below we will discuss two of these transformations: *control flow normalization* and *alias propagation*.

4.2.1 Control Flow Normalization

Control flow normalization is a transformation on programs where unstructured cycles in the flow of control are converted (or *normalised*) into structured cycles. In general, this technique is used to reduce the complexity of program transformations, automatic parallelisation, and program analysis such as data flow analysis.

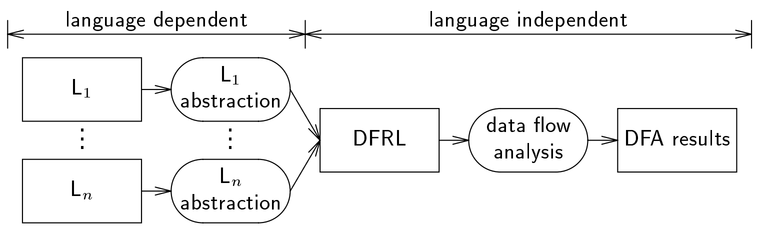


Figure 3: Separation of abstraction and data flow analysis

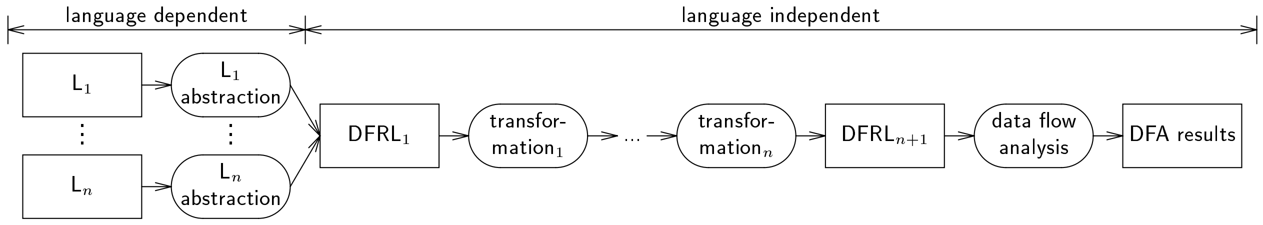


Figure 4: Transformations on the data flow representation language

Control flow normalization has several advantages for the data flow analysis process. First, it reduces the number of syntactical constructs which must be treated during analysis or transformation.

Second, the normalised program has a much simpler (denotational) semantics. On the one hand, a language with structured flow of control can be described using *direct semantics*: the semantics of control commands are simple compositions of the semantics of their components. This makes it easy to define structured analysis or transformations on such a language. On the other hand, in order to describe a language with unstructured flow of control, we must switch to the more difficult *continuation semantics*. We do no longer have compositionality of semantics which results in increased complexity in any program analysis that models this semantics (such as data flow analysis). More information on direct and continuation semantics can be found in [Sch86].

Third, the specific method of control flow normalization used in our architecture makes it unnecessary to perform interval analysis as part of the data flow analysis [Moo96]. Interval analysis is a technique to divide a (control flow) graph into sets of nodes (intervals) so that analysis of the complete graph can be composed from the analysis of the intervals. Normalisation makes this step obsolete since the structure of the program becomes so manifest in the syntax tree of the program that all interval information can be extracted directly [Amm92].

In order to support control flow normalization, we distinguish two languages in our family of data flow representation languages. Both languages have statements to denote data flow- and alias information. Furthermore, the first has statements to denote both structured and unstructured flow of control while the second is a subset that only has statements to denote structured flow of control. We will refer to the first language as DHAL (Dataflow High Abstraction Language) and to the second language as *s*DHAL (*structured* DHAL). Thus, control flow normalization is a translation from DHAL programs to *s*DHAL programs. These languages will be described in more detail in Section 5. In the meanwhile, Table 1 gives an overview of the features supported by the various languages.

4.2.2 Alias Propagation

Another example of a simplifying transformation is the so-called *alias propagation*. Aliasing occurs when there are two or more variables that refer to the same memory location. As a result, the modification of one will result in the modification of the others and vice versa. Since aliasing may cause side effects such as hidden modification of variables, it complicates almost all program analysis, including data flow analysis [LR91, MLR$^+$93]. Alias propagation is a transformation on programs where alias information is integrated with the data flow information. This allows a much simpler *alias-free* data flow analysis of the resulting code.

In order to support this process, we distinguish yet another language in our family of data flow representation languages. This third language is a subset of *s*DHAL and only has statements to denote data flow information and statements to denote structured flow of control. We will refer to this language as *p*DHAL (*propagated* DHAL since all alias information is propagated in this language). Thus, alias propagation is a translation from *s*DHAL programs to *p*DHAL programs.

5 DHAL, a Data Flow Representation Language

In this Section, we will present a data flow representation language called DHAL: *Data flow High Abstraction Language*. It has generic statements for denoting data flow-, control flow- and alias information. Our goal is to design

	data flow info	structured control flow	alias info	unstructured control flow
DHAL	∗	∗	∗	∗
*s*DHAL	∗	∗	∗	
*p*DHAL	∗	∗		

Table 1: Overview of the features of the three DHAL languages

a data flow representation language which is an easy abstraction target for imperative programming languages. To achieve this goal, our language is based on the common concepts used in those languages.

The language constructs of imperative programming languages can be divided into two types:

Elementary constructs These represent elementary operations on objects in memory. Examples of these constructs include assignments, read and write statements. In DHAL, we will replace these constructs with more abstract ones that only describe data flow information in terms of the definition or use of access paths or constructs that describe aliasing.

Control constructs These do not operate on an object but decide which piece of code is evaluated next. We call them *conditional* or predicative when this decision is made depending on some property of an object in memory. They are called *unconditional* otherwise. Examples of control constructs include *if-then* statements and the ; operator to form the sequential composition of two statements. The first is conditional and the latter is an unconditional control construct.

Since data flow analysis is a static form of program analysis, we do not consider actual (or dynamic) values of program variables. Therefore, it cannot be determined whether a condition fails or succeeds. However, in order to show the correspondence between the statements in DHAL and in the language that is abstracted from, we have chosen to resemble the standard structures for conditional control constructs.

In the rest of this section, we will take a closer look at the design and specification of our data flow representation languages. We start by describing the data types of the DHAL language, then we present the DHAL kernel language which is the basis of our data flow representation languages. Finally, we will describe the features that are added to define the three languages described in the previous Section and give a formal specification of these languages.

5.1 Data Types and Expressions

Since we do not consider actual (or dynamic) values of program variables, the DHAL languages do not contain expressions and only have a single data type: program variables that are either defined or used.

Note that we use the notion of *access paths* as an extension of basic program variables. Access paths are generic identifiers that refer to storage locations [LH88]. They can be constructed from basic program variable names, pointer indirection operators and structure field operators. For example, in the C programming language [KR88], access paths would typically include the '`*`' indirection- and the '`[ ]`', '`->`' and '`.`' field selection operators.

5.2 The DHAL Kernel Language

The DHAL kernel language is the basis of our data flow representation languages. It contains constructs to define programs, procedures and add structured flow of control. Programs consist of a program name, a list of declarations and a list of statements. Furthermore, the DHAL kernel language contains a '`no-decl`' declaration to denote an empty declarations, and a 'τ' statement to denote the *skip* or *no-operation* statement.

module Dhal-kernel-syntax
imports Variable-syntax
exports
  sorts DHALPROGRAM DHALDECL DHALDECLS DHALSTAT DHALSTATS

context-free syntax

program ID begin DHALDECLS DHALSTATS end	$\rightarrow$ DHALPROGRAM
DHALDECL+	$\rightarrow$ DHALDECLS
no-decl	$\rightarrow$ DHALDECL
τ	$\rightarrow$ DHALSTAT

5.2.1 Procedures

Procedure declarations consist of a procedure name, a list of parameters, a list of declarations and a list of statements. An advantage of using nested procedure declarations is that it allows us to do *interprogram* data flow analysis (i.e., data flow analysis on complete systems). We can translate the programs of a system to DHAL procedures (using sub-procedures for the program's procedures) and translate the job control scripts to the body of the DHAL program.

module Dhal-kernel-syntax
context-free syntax
proc ID VARLIST begin DHALDECLS DHALSTATS end $\rightarrow$ DHALDECL

5.2.2 Structured Flow of Control

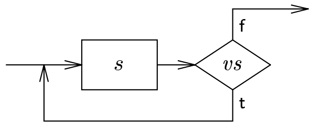
A theorem of Böhm and Jacopini [BJ66] states that any program can be expressed using only three structured control constructs: composition, conditional and iteration. All DHAL languages will have exactly these three constructs to denote structured flow of control:

Composition A series of DHAL statements is composed from two smaller series of DHAL statements using a sequential composition operator. In DHAL we use the infix operator ';' as sequential composition operator, similar to most imperative programming languages.

Conditional We use the (standard) 'if *vs* then s_1 else s_2 fi' structure for the conditional construct where s_1 and s_2 are (series of) statements and *vs* denotes the set of access paths on which the choice in the original program depended on. We will call *vs* the *dependency set* for the statements in the branches. Note that since the choice between the *then* and *else* branch cannot be made, there is no difference in DHAL between 'if *vs* then s_1 else s_2 fi' and 'if *vs* then s_2 else s_1 fi' since both mean "based on *vs* a choice is made between s_1 and s_2".

Iteration As the iteration construct, we use 'do *s* while *vs*' where *vs* is the set of access paths which are used to determine whether iteration should continue or end. There are several arguments in favour of the *do-while* variant over the *while-do* variant of iteration. The most important is *prevention of code duplication* which is shown in Examples 5.1 and 5.2.

Example 5.1 The following flow diagram can be easily translated into a *do-while* structure whereas conversion in a *while-do* structure results in code duplication:



1. do *s* while *vs*
2. *s*; while *vs* do *s* od

The main disadvantage is that we do not know the size of *s*. In the worst case *s* represents the complete program except for the last conditional construct. In that case using a *while-do* structure would result in twice as much code. □

One could argue that this duplication is prevented only because the flow diagram in the example above fits exactly on a *do-while* structure. However, the following example shows that *do-while* structures can be used also without problems to represent flow diagrams which fit naturally to *while-do* structures:

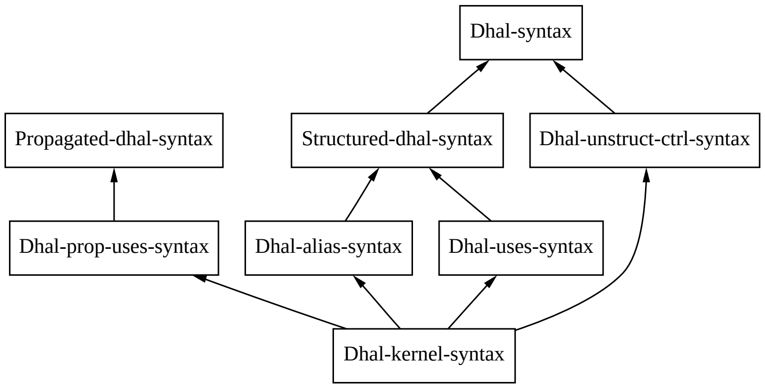
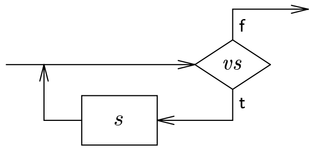


Figure 5: Import structure of DHAL, *s*DHAL and *p*DHAL.

Example 5.2 The flow diagram below is the standard translation of a *while-do* structure. As is shown, it can be converted quite easily to a *do-while* structure using an extra conditional construct:



1. while *vs* do *s* od
2. if *vs* then do *s* while *vs* fi

Although we need extra code to represent this diagram using a *do-while* structure, we know that the code growth is constant and very small since we only need one extra conditional statement. ☐

module Dhal-kernel-syntax
  context-free syntax
    {DHALSTAT ";"}+ → DHALSTATS
    if VARSET then DHALSTATS else DHALSTATS fi → DHALSTAT
    do DHALSTATS while VARSET → DHALSTAT

5.3 Language Features

New data flow representation languages can be defined by adding features to the DHAL kernel language. In this section we will describe a number of features that are added to define the three data flow representation languages described before: DHAL, *structured* DHAL (*s*DHAL) and *propagated* DHAL (*s*DHAL). The languages are defined by importing the appropriate features (see Table 1). Figure 5 presents an overview of the import structure of these languages.

5.3.1 Data Flow Information

We are interested in generic statements to describe the data flow information of pieces of code in various (imperative) programming languages. We propose to use statements which relate the set of *defined* access paths in a piece of code with the set of access paths that are *used* for that definition or with a function call.

module Dhal-uses-syntax
imports Dhal-kernel-syntax$^{5.2}$
exports
  context-free syntax
    VARSET "←" VARSET → DHALSTAT
    VARSET "←" ID VARLIST → DHALSTAT

For example, consider the following statement in a Pascal like language: $x := u + 3 * (v/u)$. This statement uses access paths u and v and defines access paths x. It can be written in DHAL as $\{x\} \leftarrow \{u,v\}$. Some other example translations are:

source code	DHAL abstraction	source code	DHAL abstraction
$x := u + 3 * (v/u)$	$\{x\} \leftarrow \{u,v\}$		$\{h_1\} \leftarrow g[u];$
$y := f(v,w)$	$\{y\} \leftarrow f[v,w]$	$p := f(g(u)) + v$	$\{h_2\} \leftarrow f[h_1];$
$print(w)$	$\{\} \leftarrow \{w\}$		$\{p\} \leftarrow \{h_2,v\}$

During the abstraction phase, the user can choose which functions are to be analysed further and which functions are ignored (e.g., analyse user-defined function f but ignore the standard *print* function). Furthermore, we propose adding temporary access paths for storage of intermediate results during analysis of more complex procedure calls.

5.3.2 Aliases

Aliasing occurs when two or more access paths refer to an overlapping storage location. As a result, the modification of one will result in the modification of the others and vice versa. In classic data flow literature (see for example [Hec77, ASU86]), the alias relation is considered an equivalence relation on access paths. In [Moo96], we show that this assumption leads to inaccurate (i.e., overly conservative) results. We propose a subdivision of the alias relations into four distinct types based on the following properties

must **or** ***maybe*** **aliases:** two access paths are considered *must* aliases at a given program point if and only if it can be determined that they refer to overlapping storage locations at that point for *all* execution paths of the program; they are considered *maybe* (or *may*) aliases otherwise.

complete **or** ***partial*** **aliases:** two access paths are considered *complete* aliases at a given program point if and only if they refer to exactly the same storage location at that point; they are considered *partial* aliases otherwise (i.e., the pair (p,q) is considered a *partial* alias if p refers to a storage location which (partially) includes the storage location that q refers to, or vice versa).

A typical example of a *complete must* alias is the pair (x,y) of two access paths that are declared equivalent using the Fortran statement EQUIVALENCE x,y. The pair $(A[i],A[j])$ of two array elements is considered a *complete may* alias at all points where it cannot be determined that i is equal to j. A trivial example of a *partial must* alias is the pair $(x,x.f)$ where x is a record and f is one of its fields. In a similar way is the pair $(A[i],A[j].f)$ considered a *partial may* alias at those points where it cannot be determined for *all* execution paths that i equals j.

Corresponding to the proposed division, we have four statements to denote the alias relations that hold at a certain program point: complete must aliases ($\leftrightarrow$), complete may aliases ($\leftrightsquigarrow$), partial must aliases ($\rightarrow$), and partial may aliases ($\rightsquigarrow$).

```
module Dhal-alias-syntax
imports Dhal-kernel-syntax^5.2
exports
  context-free syntax
    "↔" ALIASSET  → DHALSTAT
    "↭" ALIASSET → DHALSTAT
    "→" ALIASSET  → DHALSTAT
    "⇝" ALIASSET  → DHALSTAT
```

5.3.3 Propagated Data Flow Information

To improve the precision of data flow analysis after the propagation of aliases, we want to distinguish the *direct* definitions from those definitions that are *induced* by aliasing. For example, the definition of x in the statement $x := 3 * z$ is a direct definition and, when x and y are aliases, the implicit definition of y in this statement is an alias induced definition.

In order to distinguish between these two, we use special statements to denote data flow information after alias propagation. These statements separate the set of *direct* definitions and the set of *alias induced* definitions and relate them to a set of uses or a procedure call. For more detailed information on how this precision improvement is achieved, we refer to [Moo96].

module Dhal-prop-uses-syntax
imports Dhal-kernel-syntax[5.2]
exports
  context-free syntax
    VARSET "@" VARSET "←" VARSET → DHALSTAT
    VARSET "@" VARSET "←" ID VARLIST → DHALSTAT

5.3.4 Unstructured Flow of Control

To denote unstructured flow of control, we have two additional statements: 'goto' and 'label'. The label statement is used to identify a program point; the goto statement is used to let program execution continue at the corresponding program label.

module Dhal-unstruct-ctrl-syntax
imports Dhal-kernel-syntax[5.2]
exports
  context-free syntax
    goto ID → DHALSTAT
    label ID → DHALSTAT

6 Conclusions

We have developed a software architecture for the language independent specification and execution of data flow analysis. Language independency is achieved by separating the source language from the *data flow representation language*: the language on which the data flow analysis is performed. This separation has several advantages: first, all knowledge about the source language is assembled in the component that performs the abstraction from the source language to the data flow representation language. Consequently, data flow analysis algorithms can be re-used for a new source language whenever an abstraction for that language becomes available, and when the syntax of a language changes, only the component that performs the abstraction needs to be changed. Second, the results of multiple analysis can be combined very easily since they use the same format. This even holds for the results obtained by analysing completely different languages. A feature which is desirable for analysis of (legacy) systems written in multiple languages e.g, COBOL and JCL. Third, this approach allows the definition of common components that facilitate tasks that repeatedly occur in data flow analysis algorithm such as control flow normalization and alias propagation. Finally, a common model like our data flow representation language might form a suitable communication platform for information exchange on data flow analysis algorithms and their merits.

6.1 Future Work

Application of the framework requires the definition of an abstraction component for the language to be analysed. An abstraction for COBOL was already developed in [MS97]. Future work includes the definition of these abstractions for a number of other programming languages such as PL/I, JCL, C, Fortran.

Work should be done on the incorporation of alias detection (or calculation) in the framework. This detection is proven to be NP-hard or co-NP-hard in the presence of multiple levels of indirection [LR91], so techniques for the safe approximation of these alias relations should be investigated. Work in this direction has been done by [HPR89, LR92, CBC93, LRZ93]. Finally, there is a relation between detection of aliases and a relatively new technique called *points-to analysis* [EGH94] which should be investigated. Using points-to analysis, all kinds of program information can be derived, possibly including the alias relations between access paths [Ruf95, Ste96].

References

[Agr91] H. Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, Department of Computer Science, August 1991.

[Amm92] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BJ66] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL '93)*, pages 232–245. ACM, 1993.

[CC77] 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 Annual ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.

[CC92] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL '92)*, pages 84–94, January 1992.

[CR81] L. A. Clarke and D. J. Richardson. Symbolic evaluation methods for program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 9, pages 264–300. Prentice-Hall, 1981.

[DKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special Issue on Automatic Programming.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994. *SIGPLAN Notices*, 29(6).

[Har93] M. J. Harrold. Using data flow analysis for testing. Technical Report 93-112, Department of Computer Science, Clemson University, 1993.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, Amsterdam, 1977.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *SIGPLAN Notices*, 24(7):28–40, July 1989.

[JN94] N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[Ken81] K. W. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 35–46, June 1988. *SIGPLAN Notices*, 23(7).

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 93–103, January 1991.

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 235–248, 1992. *SIGPLAN Notices* 27(7).

[LRZ93] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993. *SIGPLAN Notices*, 28(6).

[MLR$^+$93] T. J. Marlowe, W. G. Landi, B. G. Ryder, J.-D. Choi, M. G. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, September 1993.

[Moo96] L. Moonen. Data Flow Analysis for Reverse Engineering. Master's thesis, University of Amsterdam, Programming Research Group, 1996. Appeared as Technical Report P9613. Available by anonymous ftp from ftp.wins.uva.nl, file pub/programming-research/MasterTheses/Moonen.ps.gz.

[MR90] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. A unified model. *Acta Informatica*, 28:121–163, 1990.

[MS97] L. Moonen and A. Sellink. COBOL Data Flow Analysis. In A. van Deursen, P. Klint, and G. Wijers, editors, *Program Analysis for System Renovation*, Resolver Release 1, chapter 11. CWI, Amsterdam, January 1997.

[OCS94] C. M. Overstreet, R. Cherinka, and R. Sparks. Using bidirectional data flow analysis to support software reuse. Technical Report TR-94-09, Old Dominion University, Computer Science Department, June 1994.

[OCTS94] C. M. Overstreet, R. Cherinka, M. Tohki, and R. Sparks. Support of software maintenance using data flow analysis. Technical Report TR-94-07, Old Dominion University, Computer Science Department, June 1994.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995. *SIGPLAN Notices*, 30(6).

[Sch86] David A. Schmidt. *Denotational Semantics*. Allyn and Bacon, Boston, 1986.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23th Annual ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 32–41. ACM Press, January 1996.