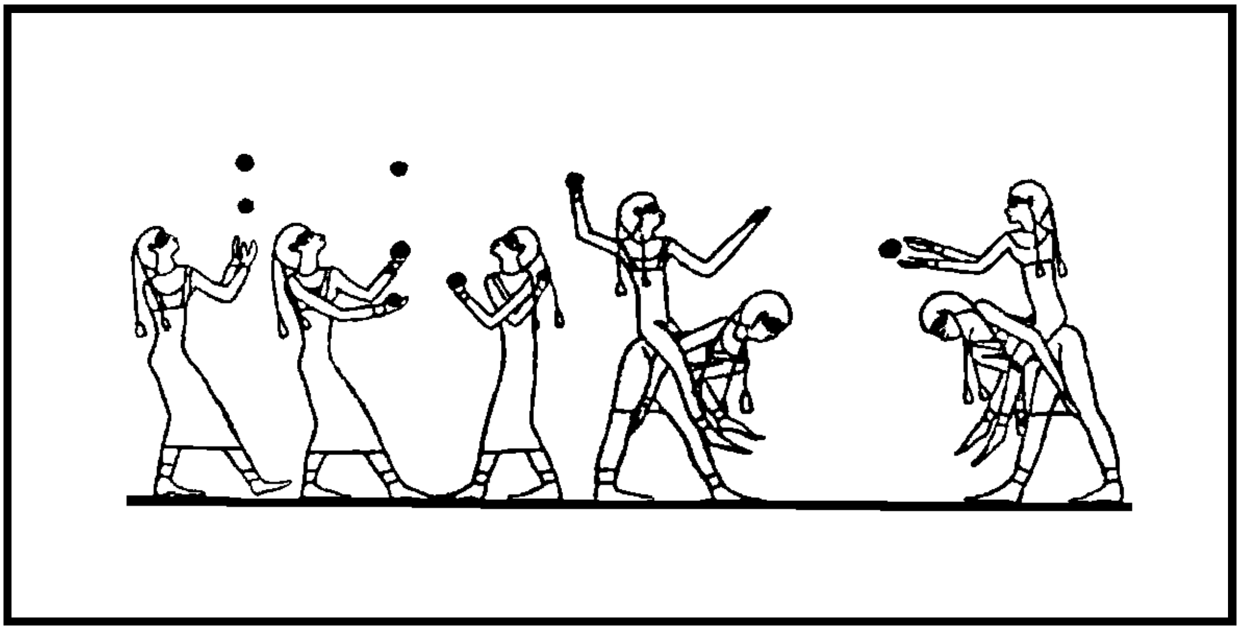
University of Amsterdam
Programming Research Group



Data Flow Analysis for Reverse Engineering

Leon Moonen

Report P9613 September 1996

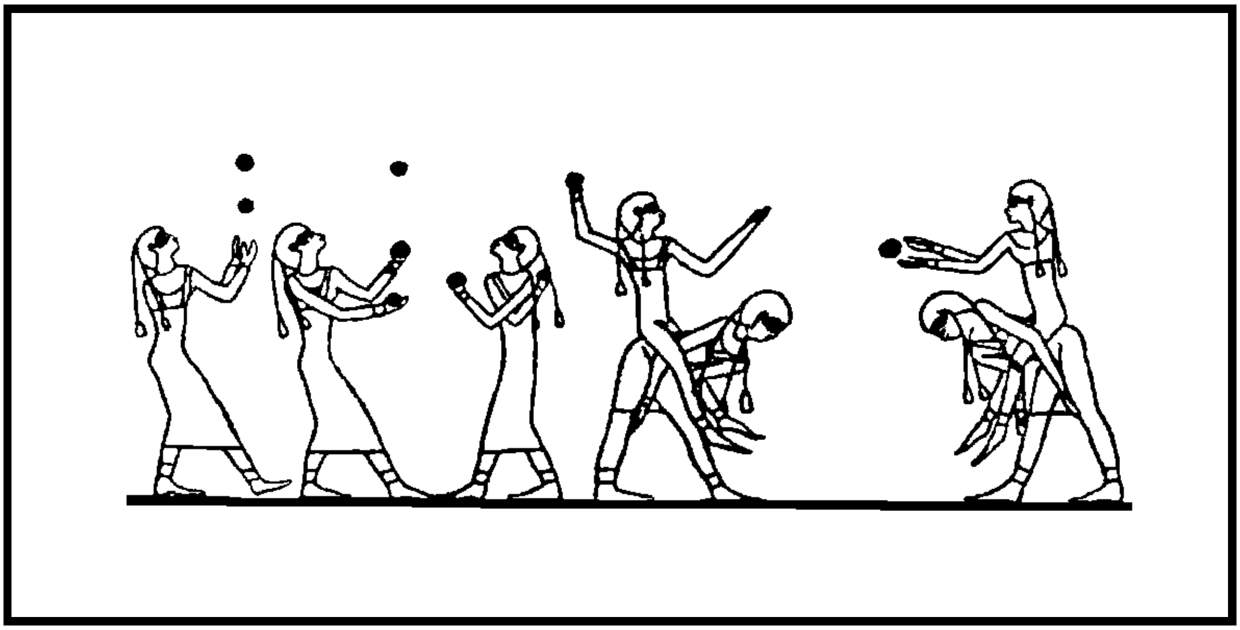
University of Amsterdam
Department of Computer Science
Programming Research Group

Data Flow Analysis for Reverse Engineering

Leon Moonen

Report P9613 September 1996

Data Flow Analysis for Reverse Engineering



Leon Moonen

Masters Thesis in Computer Science
Programming Research Group
University of Amsterdam
September 1996
Thesis supervisor: Prof. dr. P. Klint

Leon Moonen

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7582
e-mail: leon@fwi.uva.nl

COLOFON

Yet another LaTeX file ...
Figures programmed in PIC.

Cover picture: The oldest known depiction of juggling: a wall painting found in one of the crypts of the Beni Hassan tombs, dating from the middle kingdom of the ancient Egyptian civilization (between 1994 and 1781 B.C.).

Preface

This thesis is the result of my graduation research project carried out at the Programming Research Group of the Department of Computer Science, University of Amsterdam. The starting point of the project was the investigation and specification of techniques to support data flow analysis for reverse engineering and program understanding.

I would like to thank Prof. dr. P. Klint who introduced me to the project, and gave me a lot of freedom in doing things as I thought best. The members of the Programming Research Group, especially dr. M.G.J. van den Brand and dr. C. Verhoef, supported me on various aspects of my work and make a nice environment to work in. Paul Klint, Chris Verhoef, Mark van den Brand and Arie van Deursen read draft versions of this document and gave many useful remarks to improve its readability.

Finally, Ivonne, my parents, friends, and the members of the Department of Pure Juggling provided all kinds of useful distraction at times when I needed it most.

I would like to thank you all.

Leon Moonen
Amsterdam,
September 1996

Contents

List of Figures

List of Algorithms

List of ASF+SDF modules

The number between brackets after the module name indicates the section (or subsection) where that module is specified.

Chapter 1

Introduction

Data flow analysis is a process for collecting run-time information about data in programs without actually executing them. In this thesis, 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, for example, play a key role in the determination of a system's logical components and their interaction.

The general support of program understanding and reverse engineering requires the ability to analyze a variety of source languages and data flow problems. This can be established by using a framework which facilitates the specification and execution of data flow analysis algorithms. We will present a flexible and generic software architecture for describing and performing data flow analysis. All components of this framework and their interaction are algebraically specified in ASF+SDF.

1.1 Data flow analysis

Data flow analysis is a process for collecting run-time information about a program without actually executing that program. For example, consider the following program statement: $x := y + z$. Data flow analysis can be used to answer questions like: "What definitions of y might have set the value of y used here?", "On which variables does the definition of x depend indirectly?" and "Where will the value of x computed here, be used in the program?".

The properties that can be found usually apply to all possible execution paths and therefore give global information. This information can be used in a wide range of applications. Examples include the support of the optimization phase in compilers [Hec77, ASU86], automated debugging [Agr91], improvement of software reliability [FO76], program testing [Har93], and program development and maintenance [OCTS94, OCS94].

This thesis focuses at the use of data flow analysis to support program understanding and reverse engineering. *Reverse engineering* can be seen as the inverse of the conventional software engineering model: it is a process which analyses the implementation of a software system in order to distinguish its logical components and their interaction and represent the system at a higher level of abstraction. An overview of this area is geven in [BKV96]. Data flow analysis is useful for such applications since the information obtained can be used to compute relationships between data objects in programs. These relations play a key role in the determination of a system's logical components and their interaction. At a lower level

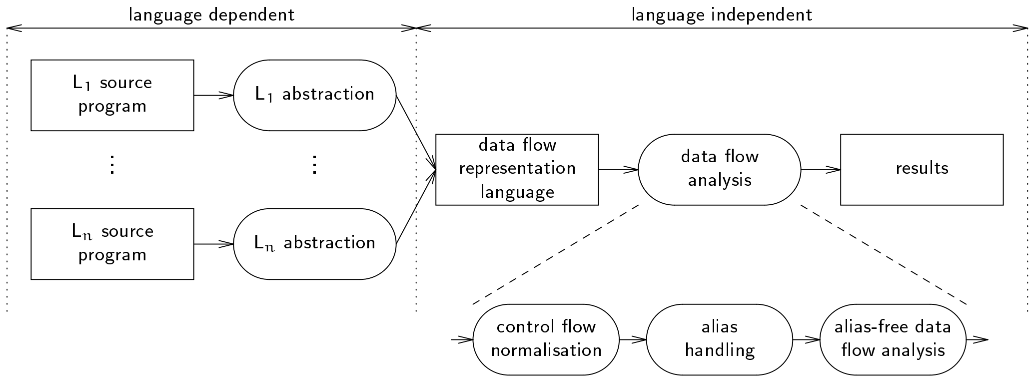


Figure 1.1: Overview of the proposed framework

of abstraction, even more information can be derived using data flow analysis: procedural dependencies, dependencies between variables and data modified by given pieces of code.

Reverse engineering and program understanding are, for example, very important for testing large legacy systems on their year 2000 compliance and for code conversions that are necessary for the introduction of the Euro single currency unit in financial software. Both are huge problems which occur frequently in software systems. These problems are not language specific, they can appear in systems written in any of the hundreds of different programming languages that are available. A language independent approach to data flow analysis as presented in this thesis is a first step to a systematic solution of these problems.

1.2 A generic framework for data flow analysis

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

- duplication of algorithm implementation efforts;
- whenever the language to be analyzed changes, all algorithm implementations need to be redone. We expect this problem to grow in the near future due to the increased use of special purpose languages with relatively short life cycles;
- information exchange between researchers and/or software practicioners is cumbersome due to language specific approaches.

To avoid these problems, we propose a flexible and generic framework that supports the specification and execution of data flow analysis. The framework allows language independent specification of data flow analysis algorithms by separating the source language from the *data flow representation language*: the language on which the data flow analysis is performed. Furthermore, it is built using a number of generic components that facilitate common features found in data flow analysis. Such a framework facilitates rapid prototyping of data flow analysis algorithms and it can be used for the analysis of a wide range of source languages (see Figure 1.1).

This thesis contains the formal specification of the proposed data flow representation language. Furthermore, we specify the language independent components that support the two following features:

- *control flow normalization*: a transformation process on programs where unstructured cycles in the flow of control are converted (or *normalized*) into structured cycles.
- *alias handling*: aliasing occurs when there are two or more identifiers that refer to the same memory location. We describe a technique for precise propagation of alias information in programs. Such propagation allows *alias-free* data flow analysis of the resulting code.

1.2.1 Control flow normalization

Control flow normalization (or *goto elimination*) is a transformation process on a program where unstructured cycles in the flow of control are converted (or *normalized*) into structured cycles. Control flow normalization is based on a theorem by Böhm and Jacopini [BJ66] which states that every program P can be transformed into another equivalent program P' which only uses the three control structures: composition, conditional and iteration. Ashcroft and Manna described the first algorithm for control flow normalization [AM72] and a lot of work has been done since then (e.g. [WO75, Bak77, AKPW83]). All these techniques convert programs with an unstructured flow of control into structured programs whose control flow graph is reducible. The control flow normalization technique that is used in this thesis was developed by Ammarguellat [Amm92]. It generates programs whose control flow graphs satisfy a condition that is stronger than reducibility and therefore the normalized programs are simpler in structure and syntax. Moreover, while some of the previous techniques result in a lot of code duplication, this technique only requires code duplication to solve irreducible flow graphs, a condition which does not occur very often, even in unstructured programs [Hec77].

1.2.2 Aliasing

Aliasing occurs when there are two or more identifiers that refer to the same memory location. Thus, 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 [ADS91, LR91, MLR$^+$93]. Research has been done on both detection of aliasing [CBC93, LR92, WL95] and the integration of alias information with program analysis [BC92, LRZ93, Wei80].

In traditional data flow analysis literature (see for example [Hec77, ASU86]), the alias relation is considered an equivalence relation on access paths. We show that this assumption is incorrect and leads to inaccurate (but conservative) results. We propose a subdivision of the alias relations into four distinct types and prove that only one of these is an equivalence relation. Furthermore, we describe a careful propagation of these distinct relations which results in a more precise side effect analysis.

Chapter 2

Data Flow Analysis

Data flow analysis can be described as a process to collect information about the use, definition, and dependencies of data in a program. In this chapter, we will give an introduction to data flow analysis, describe some key notions that are used in this field and discuss a few classical data flow problems. Finally, we will present our point of view: the use of data flow analysis to support program understanding and reverse engineering. This application area was already acknowledged in [Hec77] although the main focus of that book (and of much subsequent research as well) is the use of data flow analysis to support code improvement in compilers.

2.1 Introduction to data flow analysis

Data flow analysis is used 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 y used here?", "On which variables does the definition of x depend indirectly?" and "Where will the value of x computed here be used in the program?".

One can abstract from the specific variable in these questions and ask this kind of information for all variables in the program. Such general questions are called *data flow problems*. The solution of data flow problems is described using so-called *data flow analysis algorithms*. These algorithms perform data flow analysis on programs or procedures: they capture program information and compute the derived (data flow) information which is the solution to a data flow problem.

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 problem. A typical example of such a data flow equation is $\mathsf{out}(n) = \mathsf{gen}(n) \cup (\mathsf{in}(n) - \mathsf{kill}(n))$ 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 killed in n". In general, $\mathsf{in}(n)$ is a function that collects information from a subset of the neighbors of n, usually either the successors or predecessors of n in the flow graph.

Data flow analysis is a *static* form of analysis: program information is collected without actual execution of the program. Of course, in such a case it is impossible 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 makes approximations of this behavior, for example by considering both branches of an *if-then-else* statement and performing a fixed-point computation for the body of a *while* statement. A fixed-point exists since the data flow equations compute sets of identifiers or sets of expressions and we only consider programs with a finite number of identifiers and statements and therefore also only a finite number of expressions.

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 problems. Given a data flow problem, such an approximate solution can err by *overdetermination*: allowing conclusions which are stronger than actually valid, or it can err by *underdetermination*: only allowing conclusions which are weaker than actually valid. Obviously, we prefer solutions that are valid if imprecise, these are called *safe* or *conservative* solutions.

Original applications of data flow analysis come from the area of compiler optimization 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 thesis, we will focus at the use of data flow analysis to support program understanding and reverse engineering. This will have some influence on choices that are made but we will try to keep the techniques presented here as general as possible.

2.2 Data flow preliminaries

Before describing some standard data flow problems, we introduce the most commonly used notions of this field. A more detailed description of these concepts can be found in [Hec77, ASU86] which both contain good introductions to data flow analysis and its application to support code improvement in compilers.

Control flow graphs are frequently used for the description of data flow analysis. A (control) flow graph is a triple $\mathcal{F} = (N, E, \rho)$ where $\mathcal{G} = (N, E)$ is a *directed graph* consisting of a set of nodes N that represent statements, basic blocks, or procedures of a program and a set of edges E that represent the control transfers between code pieces represented by these nodes. There is an edge from node n to node m if there is an execution trace possible in which m can follow n (control can pass from n to m). A *path* in a control flow graph is a finite sequence of nodes $(n_1, n_2, \ldots, n_k)$ such that for each $i = 1, 2, \ldots, k-1$ there is an edge from n_i to n_{i+1}. In a control flow graph, there is a path from the *initial* (or *start*) node $\rho \in N$ to every node $n \in N$.

Program information is attached to a control flow graph using a labeling of the nodes and/or edges of the graph. Let $\mathcal{G} = (N, E)$ be a graph and L a set of labels. A *node labeling* of $\mathcal{G}$ is a function $f : N \to L$ which maps each node to an element of L. Similarly an *edge labeling* of $\mathcal{G}$ 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.

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 $\mathsf{def}(n)$ 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 $\mathsf{pres}(n)$ 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 $\mathsf{use}(n)$ 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 $\mathsf{nodef}(n)$ be the set of locally preserved variables of node n. Note that $\mathsf{nodef}(n)$ are exactly those variables which have a definition in $\mathsf{pres}(n)$.

2.3 Typical data flow problems

In this section, we will describe some classical data flow 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 problems, we assume that all necessary local data flow information is available in the programs control flow graph. For each data flow problem that is described, we will present a brief example of its application. More information on these applications can be found in [ASU86].

2.3.1 Reaching definitions

The *reaching definitions* at the top (respectively: bottom) of a node n are the variable definitions that can reach the top (respectively: bottom) of n. That is, the set of definitions that are preserved along the path from their definition to the top (respectively: bottom) 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.

Reaching definitions can be computed using the equations in Figure 2.1. The definitions reaching the top of a node n are all definitions reaching the bottom of predecessors of n (or none if n has no predecessors). The definitions reaching the bottom of a node n are the definitions that reach the top of n and are preserved in n or the definitions that are locally generated in n.

For each node n,

$$\mathsf{rdtop}(n) = \begin{cases} \emptyset & \text{when } \mathsf{pred}(n) = \emptyset \\ \displaystyle\bigcup_{m \in \mathsf{pred}(n)} \mathsf{rdbot}(m) & \text{otherwise} \end{cases} \tag{RD1}$$

For each node n,

$$\mathsf{rdbot}(n) = \big[\mathsf{rdtop}(n) \cap \mathsf{pres}(n)\big] \cup \mathsf{def}(n) \tag{RD2}$$

Figure 2.1: Data flow equations for reaching definitions

Note that $\mathsf{rdtop}(n)$ and $\mathsf{rdbot}(n)$ can each be described without reference to each other by making the appropriate substitutions, as is shown in Figures 2.2 and 2.3:

For each node n,

$$\mathsf{rdtop}(n) = \begin{cases} \emptyset & \text{when } \mathsf{pred}(n) = \emptyset \\ \displaystyle\bigcup_{m \in \mathsf{pred}(n)} \Big[\big[\mathsf{rdtop}(m) \cap \mathsf{pres}(m)\big] \cup \mathsf{def}(m)\Big] & \text{otherwise} \end{cases} \tag{RD3}$$

Figure 2.2: Data flow equations for definitions reaching the top

For each node n,

$$\mathsf{rdbot}(n) = \begin{cases} \mathsf{def}(n) & \text{when } \mathsf{pred}(n) = \emptyset \\ \Big[\Big[\displaystyle\bigcup_{m \in \mathsf{pred}(n)} \mathsf{rdbot}(m)\Big] \cap \mathsf{pres}(n)\Big] \cup \mathsf{def}(n) & \text{otherwise} \end{cases} \tag{RD4}$$

Figure 2.3: Data flow equations for definitions reaching the bottom

2.3.2 Live variables

The *live variables* at the top (respectively: bottom) of a node n are the sets of variables that have a definition-clear path from the top (respectively: 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 allocation* determines in which registers those values should reside.

The sets of live variables can be computed using the equations in Figures 2.4–2.6. The variables that are alive at the bottom of a node n are all variables that are alive at the top of successors of n (or none if n has no successors). The variables that are alive at the top of a node n are the variables that are alive at the bottom of n and are not defined in n or the variables that are locally used in n. Note that the equations for the live variables problem are so-called *bottom-up* equations whereas the equations for the reaching definitions problem were so-called *top-down* equations.

For each node n,

$$\mathsf{lvbot}(n) = \begin{cases} \emptyset & \text{when } \mathsf{succ}(n) = \emptyset \\ \bigcup\limits_{m \in \mathsf{succ}(n)} \mathsf{lvtop}(m) & \text{otherwise} \end{cases} \tag{LV1}$$

For each node n,

$$\mathsf{lvtop}(n) = \big[\mathsf{lvbot}(n) \cap \mathsf{nodef}(n)\big] \cup \mathsf{use}(n) \tag{LV2}$$

Figure 2.4: Data flow equations for live variables

For each node n,

$$\mathsf{lvbot}(n) = \begin{cases} \emptyset & \text{when } \mathsf{succ}(n) = \emptyset \\ \bigcup\limits_{m \in \mathsf{succ}(n)} \Big[\big[\mathsf{lvbot}(m) \cap \mathsf{nodef}(m)\big] \cup \mathsf{use}(m)\Big] & \text{otherwise} \end{cases} \tag{LV3}$$

Figure 2.5: Data flow equations for live variables at the bottom

For each node n,

$$\mathsf{lvtop}(n) = \begin{cases} \mathsf{use}(n) & \text{when } \mathsf{succ}(n) = \emptyset \\ \Big[\Big[\bigcup\limits_{m \in \mathsf{succ}(n)} \mathsf{lvtop}(m)\Big] \cap \mathsf{nodef}(n)\Big] \cup \mathsf{use}(n) & \text{otherwise} \end{cases} \tag{LV4}$$

Figure 2.6: Data flow equations for live variables at the top

2.3.3 Live definitions

The *live definitions* at the top (respectively: bottom) of a node n are the definitions that reach the top (respectively: bottom) of n and can potentially be used, i.e. all variables that are used by the definitions are alive at the top (respectively: bottom) of n. Live definitions can be used to support common subexpression elimination. Their computation is described in Figure 2.7.

For each node n,

$$\mathsf{ldtop}(n) = \{\, d \mid d \in \mathsf{rdtop}(n) \wedge \mathsf{uv}(d) \subseteq \mathsf{lvtop}(n) \,\} \tag{LD1}$$

For each node n,

$$\mathsf{ldbot}(n) = \{\, d \mid d \in \mathsf{rdbot}(n) \wedge \mathsf{uv}(d) \subseteq \mathsf{lvbot}(n) \,\} \tag{LD2}$$

where $\mathsf{uv}(d)$ denotes the set of variables that are used in definition d.

Figure 2.7: Data flow equations for live definitions

2.4 Data flow analysis for reverse engineering

The conventional software engineering model moves from a high level abstract design using stepwise refinement to a low level implementation. Nowadays, this is also called forward engineering. *Reverse engineering* can be seen as the inverse of this process: it is a process which analyses the implementation of a software system in order to distinguish its logical components and their interaction and represent the system on a higher level of abstraction. We will also refer to this technique using the phrase *program understanding*.

Data flow analysis can be useful for program understanding and reverse engineering since it can be used to compute relationships between data objects in programs. This information plays a key role for the determination of a system's logical components and their interaction. On a lower abstraction level, even more information can be derived using data flow analysis: procedural dependencies, dependencies between variables and data modified by given pieces of code.

2.4.1 Definition-use chaining

A data flow problem which is useful for the support of program understanding and reverse engineering is known as *definition-use chaining*. Suppose we have the following information for each node $n \in N$:

- $\mathsf{use}(n)$ the locally used variables of n;
- $\mathsf{def}(n)$ the locally generated definitions of n;
- $\mathsf{rdtop}(n)$ the definitions reaching the top of n;
- $\mathsf{lvbot}(n)$ the variables that are alive at the bottom of n.

By considering both $\mathsf{rdtop}(n)$ and $\mathsf{use}(n)$ we can relate each use of a variable in a node to zero or more definitions reaching that node's top. Similarly, considering both $\mathsf{lvbot}(n)$ and $\mathsf{def}(n)$ 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 2.4.1 In this example we present a small program in pseudo code and its control flow graph, annotated with the definition-use chains for all variables. Consider the following program that computes the factorial of the value in variable input and puts it in variable output:

```
program factorial
    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
```

The next figure presents the control flow graph of this program, annotated with definition-use chaining for all variables. The characters 't' and 'f' denote the path that is taken when the condition of the while is respectively true or false.

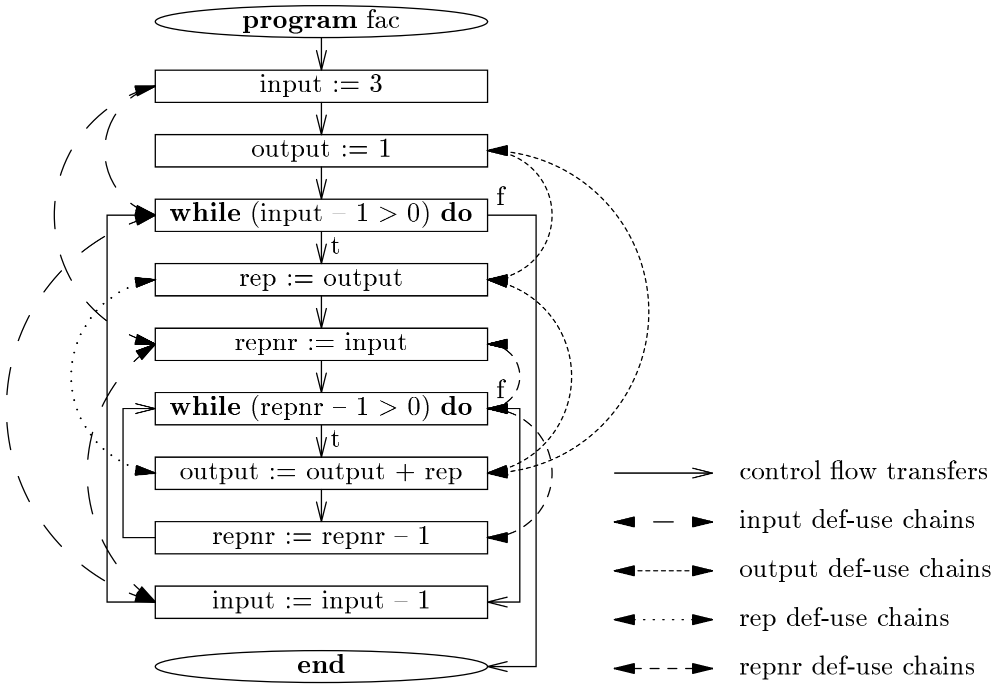


Figure 2.8: Control flow graph annotated with definition-use chains

□

2.4.2 Variable dependence tracking

Variable dependence tracking is another data flow problem that can be used for the support of program understanding and reverse engineering. It gives answers to questions like: "Given a variable ν, what other variables are either used to define the value of ν or have a value which depends in some way on ν?".

Variable dependence tracking is a so-called *closure problem*. Therefore, the data flow analysis algorithm to compute these dependencies will, in general, not find the solution in a single computing pass over the source code but needs to perform a *fixed point* computation. This means that the algorithm iteratively propagates the dependence information over the code until the variable dependencies are converged to a stable solution.

An algorithm for finding these variable dependencies will be presented in Chapter 7 where it will serve as an example of a data flow analysis algorithm that is specified using the data flow framework proposed in the next chapter.

Chapter 3

A generic framework for data flow analysis

Data flow analysis can be used to solve various program analysis problems as was shown in the previous chapter. In order to facilitate such diverse analyses, it is very useful to have tools that support the specification and execution of data flow analysis. We present a flexible and generic framework for describing and performing data flow analysis. This framework facilitates rapid prototyping of data flow analysis algorithms and allows these algorithms to be used for the analysis of a huge range of source languages.

3.1 Motivation

In general, the algorithms that solve data flow analysis problems are coded by hand and directed towards finding a solution for a specific programming language. Obviously, this practice has some undesirable consequences:

- It leads to the duplication of a considerable amount of "standard" programming work. For example, a lot of data flow analysis algorithms are based on *interval analysis* and there are serious expenses related with the repeated – programming language specific – implementation of such algorithms [ASU86]. 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 could not be analyzed in a syntax-directed manner immediately [Hec77, ASU86].

- When the programming language to be analyzed is changed, all data flow analysis algorithms have to be retargeted to analyze that new language. We expect this problem to grow in the near future due to the increased use of special purpose languages. These languages have a relatively short life cycle as opposed to the more traditional general purpose languages. For example, the high level language ESL for describing graphical user interfaces, communications and database access in client/server applications [Eas94] is no longer supported by its developers and currently, code written in ESL is reverse engineered and ported to new platforms. However, the use of ESL specific analysis algorithms in this process is questionable since such algorithms need to be retargeted when the new language is superseded again.

- Due to the language specific approach, it is very hard to describe a newly developed technique and the merits in a generic way. Therefore, information interchange on data flow analysis algorithms is cumbersome, and researchers or software engineers are regularly forced to re-invent the wheel for more complex analysis problems.

Consequently, the costs of implementing data flow analysis algorithms can prevent people from trying to develop alternative algorithms or from using other source languages.

We propose to solve these problems by means of a flexible and generic software architecture that supports the specification and execution of data flow analysis. This architecture makes use of generic components that facilitate common features found in data flow analysis. We have identified these features using small case studies and by examination of existing data flow analysis algorithms [Ken81, MR90]. The architecture furthermore separates the source language from the *data flow representation language*: the language on which the data flow analysis is performed. This has two advantages: first it allows one to write flexible data flow algorithms for a huge range of source languages (i.e. all languages that can be translated to our data flow representation language). Second, in our opinion, a data flow representation language similar to ours might become a suitable communication platform for information exchange on data flow analysis algorithms.

An additional advantage of the proposed architecture is that developing data flow algorithms becomes so straightforward that two (or more) data flow algorithms can easily be combined into a single algorithm. This improves the global efficiency of the program analysis since only a single run is needed instead of separate runs for each (sub-)algorithm followed by a run to combine the results. Furthermore, for some combinations of algorithms, the precision of the overall solution may be improved since they can now make use each others intermediate results.

3.2 Separate abstraction and data flow analysis

The algorithms in most data flow analysis software are aimed at the analysis of a specific programming language [Hec77, Ken81, ASU86]. Generally, such software is implemented using some form of abstract interpretation [CR81, CC92, JN94]. *Abstract interpretation* (or *symbolic execution*) is a technique for analyzing programs in a given language by evaluating them with a non-standard semantics for that language [CC77, AH87]. We will refer to the semantics to calculate this data flow information as *data flow semantics*. Since this is a language specific approach, such data flow analysis software is unsuitable for the analysis of other languages. However, in general, we want to be able to apply the same data flow analysis algorithms to different languages since we want to extract equivalent information.

We propose to separate the abstraction and the data flow analysis by introducing a

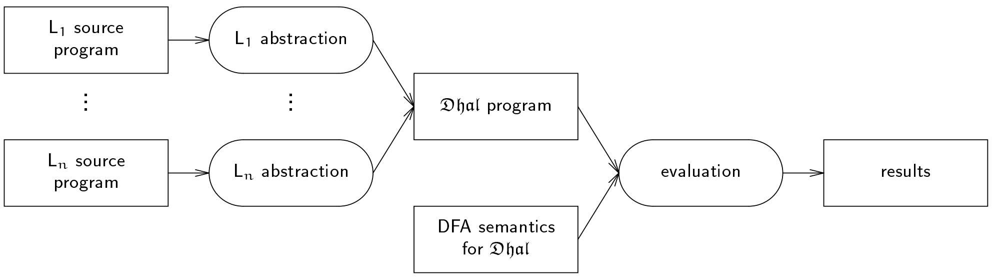


Figure 3.1: Separation of abstraction and data flow analysis

generic intermediate level. The execution of data flow analysis then consists of two consecutive steps (see also Figure 3.1):

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. Our data flow representation language is called $\mathfrak{Dhal}$: *Data flow High Abstraction Language*. It has generic statements for denoting data flow-, control flow- and alias information and is primarily designed as an abstraction language for imperative programming languages i.e. its basic programming constructs are chosen to reflect the basic constructs of these languages.

2. Perform data flow analysis at the level of the data flow representation language by evaluating $\mathfrak{Dhal}$ programs with data flow semantics.

The advantage of this approach is that all knowledge about the source language is assembled in the component that performs the abstraction and data flow algorithms can be re-used for a new source language when an abstraction for that language becomes available.

We can follow this abstarction approach since we use data flow analysis for the support of program understanding and reverse engineering. Consequently, we do not need to know all the specific program information that is required for traditional data flow analysis applications such as compiler optimization. The results of the analysis can be linked back to the original source code using *origin tracking*: an annotation technique which defines relations between pieces of code in the translated program and their counterparts 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 development environment [Kli93].

3.3 Common features

Using some small case studies and analysis of existing data flow algorithms, we have identified a number of common features that occur in data flow analysis software. These features include interval analysis, control flow normalization and alias handling. From a software engineering point of view, it is best to separate these common features from the actual data flow analysis process since this allows them to be re-used for other analysis processes.

We propose a software architecture for performing data flow analysis which uses generic components to facilitate these common features. We will give a formal specification of the support of the following features:

- *control flow normalization*
- *alias handling*

The support of *interval analysis* is not specified since we choose a specific technique for performing control flow normalization which makes it unnecessary to perform interval analysis as part of the data flow analysis process.

3.3.1 Control flow normalization

Control flow normalization is a transformation process on programs where unstructured cycles in the flow of control are converted into structured cycles. In order to support this process, we distinguish two sub-languages in our data flow representation language. Both

Figure 3.2: Common features in data flow analysis

languages have statements to denote data flow- and alias information. Furthermore, the first language has statements to denote both structured and unstructured flow of control while the second only has statements to denote structured flow of control. We will refer to the first language as $\mathfrak{Dhal}$ and to the second language as s$\mathfrak{Dhal}$ (*structured* $\mathfrak{Dhal}$). Thus, control flow normalization is a translation from $\mathfrak{Dhal}$ programs to s$\mathfrak{Dhal}$ programs (see Figure 3.2).

3.3.2 Alias handling

Alias handling is a transformation process on programs where alias information is integrated with the data flow information. In order to support this process, we distinguish yet another sub-language in our data flow representation language. This third sub-language only has statements to denote data flow information and statements to denote structured flow of control. We will refer to this language as p$\mathfrak{Dhal}$ (*propagated* $\mathfrak{Dhal}$ since all alias information is propagated in this language). Thus, alias handling is a translation from s$\mathfrak{Dhal}$ programs to p$\mathfrak{Dhal}$ programs (see Figure 3.2).

3.4 Overview of the proposed framework

Table 3.1 gives an overview of the language features that are available in the different sub-languages of our data flow representation language. Figure 3.3 presents a complete overview of the proposed framework. First, we abstract from the program in a given source language to a $\mathfrak{Dhal}$ program, then we perform control flow normalization to obtain an s$\mathfrak{Dhal}$ program, followed by alias handling to get a p$\mathfrak{Dhal}$ program. Finally, the p$\mathfrak{Dhal}$ program is evaluated using a particular data flow semantics to get the solution of the corresponding data flow problem. In the following chapters we will give a more detailed description of our data flow representation language and its sub-languages (Chapter 4), present a formal specification in ASF+SDF of the control flow normalization process (Chapter 5) and of the alias handling process (Chapter 6) and describe an example application of our framework (Chapter 7).

	data flow info	structured control flow	alias info	unstructured control flow
$\mathfrak{Dhal}$	*	*	*	*
s$\mathfrak{Dhal}$	*	*	*	
p$\mathfrak{Dhal}$	*	*		

Table 3.1: Overview of the features of the data flow representation languages

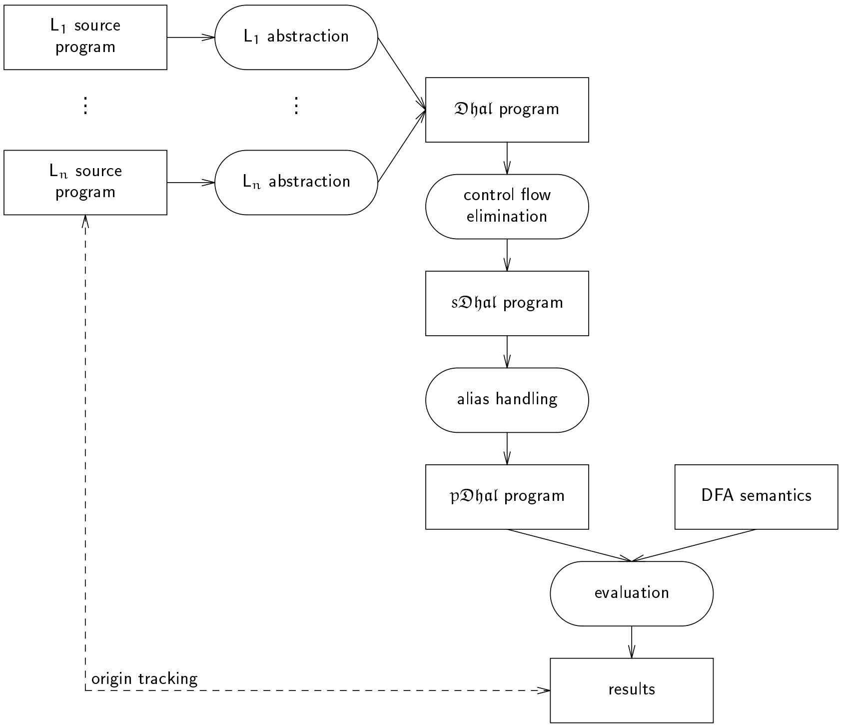


Figure 3.3: Complete overview of the proposed software architecture

Chapter 4

Dhal, a Data flow High Abstraction Language

"Maybe we can eventually make language a complete impediment to understanding."
– Hobbes

In this chapter, we will take a closer look at the design and formal specification of our data flow representation language(s). As described before, we distinguish three distinct languages depending on the language features that are available. We will give a modular specification of these features in ASF+SDF, motivate their appearance and finally, we will give a formal specification of the three languages that use them.

4.1 Data types

Data flow analysis is a static form of program analysis, i.e. during data flow analysis we do not consider actual (or dynamic) values of program variables but only the fact whether or not a particular variable has been defined and which variables are used for that definition.

We propose to use the notion of *access paths* as an extension of basic program variables. Access paths are identifiers that refer to storage locations [LH88]:

Definition 4.1 (Access path) An access path is a left-value expression that refers to a storage location and is constructed from basic program variable names, pointer indirection operators and structure field operators. □

For example, in the C programming language [KR78], access paths would typically include the '`*`' indirection- and the '`[ ]`', '`->`' and '`.`' field selection operators.

Identifiers are the basic data type in our data flow representation language. They are used to denote access paths and program or function names. Furthermore, there are three compound data types: sets of identifiers which are used to describe side-effects, lists of identifiers which are used for function parameters, and sets of identifier tuples, which are used to describe alias relations. These data types are formally specified in appendices C.3 and C.4.

4.2 Language structure

We want our data flow representation language to be an easy abstraction target for imperative programming languages. Therefore, it resembles a great deal of the structure of such languages. 𝔇𝔥𝔞𝔩 programs have an identifier, start with zero or more (procedure) declarations followed by a (series of) statements. The 'τ' statement denotes the *skip* or *no-operation* statement.

module Dhal-basic-syntax
imports Layout[C.1] Identifiers[C.3] Identifier-Containers[C.4]
exports
  sorts DHALPROGRAM DHALDECLS DHALDECL DHALSTAT
  context-free syntax
    program ID "{" DHALDECLS DHALSTAT "}" → DHALPROGRAM
    DHALDECL* → DHALDECLS
    τ → DHALSTAT
  variables
    δ $[0-9\,']*$ → DHALDECL
    "δ^{+}"$[0-9\,']*$ → DHALDECL+
    "δ^{*}"$[0-9\,']*$ → DHALDECL*
    σ $[0-9\,']*$ → DHALSTAT

4.2.1 Procedures

Procedure declarations consist of a procedure name, a list of parameters and a body.

module Dhal-proc-syntax
imports Dhal-basic-syntax[4.2]
exports
  context-free syntax
    proc ID LIST[ID] "{" DHALSTAT "}" → DHALDECL

We do not allow nested procedure declarations inside a procedure declaration since it would result in all kinds of scope checking during data flow analysis. The disadvantage is that abstraction from Pascal like languages, which do have nested procedure definitions, becomes a little harder. We propose to abstract from such nested procedures by flattening their definitions and use structured names for their identification. For example, we use the name $f.g.h(x, y)$ for a procedure h which was declared within procedure g which was declared within procedure f.

4.3 Language features

The language constructs that are the building blocks of an imperative programming language 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.

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 latter is an example of an unconditional control construct.

Since our data flow representation language is designed for abstraction from imperative programming languages, it also incorporates both these types.

4.3.1 Elementary constructs

We distinguish two kinds of elementary constructs: first, statements that can be used to describe the side effects of a piece of code in the original program which is to be analyzed, and second, statements that can be used to describe aliasing between two or more access paths.

To improve the precision of data flow analysis, we describe the side effects after alias propagation using special case of side effect statements called *propagated side effects*. The motivation for this approach is described more detailed in Chapter 6.

Side effects

We are interested in generic statements to describe the side effects 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-side-effect-syntax
imports Dhal-basic-syntax$^{4.2}$
exports
  context-free syntax
    SET[ID] "⋉" SET[ID] → DHALSTAT
    SET[ID] "⋉" ID LIST[ID] → 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 denoted in $\mathfrak{Dhal}$ as $\{x\} \ltimes \{u, v\}$. Some other example translations are:

source code	$\mathfrak{Dhal}$ abstraction
$x := u + 3 * (v/u)$	$\{x\} \ltimes \{u, v\}$
$y := f(v, w)$	$\{y\} \ltimes f[v, w]$
$z := sin(d)$	$\{z\} \ltimes \{d\}$
$write(w)$	$\{\} \ltimes \{w\}$
$p := f(g(u)) + v$	$\left\{\begin{array}{l}\{h_1\} \ltimes g[u];\\ \{h_2\} \ltimes f[h_1];\\ \{p\} \ltimes \{h_2, v\}\end{array}\right.$

As is shown in the examples, during the abstraction the user can choose which functions are to be analyzed further and which functions are ignored (e.g. analyze user-defined functions but ignore the standard library functions). Furthermore, we propose adding temporary variables for storage of intermediate results during analysis of more complex procedure calls.

Aliases

Aliasing occurs when two or more access paths, at the same point in a program $\mathcal{P}$, refer to an overlapping storage location. We distinguish four different types of aliasing:

complete must aliases ($\leftrightarrow$)	*complete may* aliases ($\leftrightsquigarrow$)
partial must aliases ($\rightarrow$)	*partial may* aliases ($\rightsquigarrow$)

The precise definition and motivation for this division is presented in Chapter 6. For now, it is sufficient to know that there are four types of aliases in order to explain the syntax. Corresponding to the proposed division, we have four statements to denote the alias relations that hold at a certain program point.

module Dhal-alias-syntax
imports Dhal-basic-syntax[4.2]
exports
context-free syntax
"$\leftrightarrow$" SET[ID×ID] $\rightarrow$ DHALSTAT
"$\leftrightsquigarrow$" SET[ID×ID] $\rightarrow$ DHALSTAT
"$\rightarrow$" SET[ID×ID] $\rightarrow$ DHALSTAT
"$\rightsquigarrow$" SET[ID×ID] $\rightarrow$ DHALSTAT

One could also think of using a more incremental approach for describing the alias relations that hold at given program points. This approach requires statements that add, and statements that remove tuples from the different alias relations. Unfortunately, there are some drawbacks to this incremental approach. In general, programming languages tend to have statements that allow us to determine alias definitions but not "unalias" definitions in a program. Therefore, it is unclear how to determine the tuples that need to be removed from the alias relations as the result of a given piece of code. Also, the algorithms for handling aliases using an incremental approach are of a considerable higher complexity than the algorithms using the non-incremental approach which will slow down the processing of $\mathfrak{Dhal}$. For these reasons, we choose to denote the full alias relation at the points were they change. The disadvantage of this approach is that it increases the size of the $\mathfrak{Dhal}$ programs. However, this is a minor disadvantage since it is not an increment in the number of statements but in the size of some of the statements and we feel that it will have no considerable impact on the processing of $\mathfrak{Dhal}$.

Propagated side effects

After the propagation of aliases into the side-effect statements, we want to distinguish the *direct* definitions from those definitions that are *induced* by aliasing. For this reason, we need to revise the statements to describe side-effects since we need to relate two sets of definitions to a set of uses or a procedure call.

module Dhal-prop-side-effect-syntax
imports Dhal-basic-syntax[4.2]
exports
context-free syntax
"(" SET[ID] "," SET[ID] ")" "$\ltimes$" SET[ID] $\rightarrow$ DHALSTAT
"(" SET[ID] "," SET[ID] ")" "$\ltimes$" ID LIST[ID] $\rightarrow$ DHALSTAT

4.3.2 Control constructs

Structured flow of control

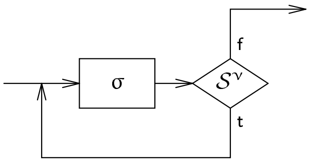
A theorem of Böhm and Jacopini [BJ66] states that any program can be expressed using only three structured control constructs. These control constructs are: composition, conditional and iteration. Our data flow representation language will have exactly these three constructs to denote structured flow of control.

Composition As in most programming languages, $\mathfrak{Dhal}$ uses the infix operator '**;**' as the sequential composition operator.

Conditional As the conditional statement, we have '**if** $\mathcal{S}^\vee$ **then** $\{\sigma_1\}$ **else** $\{\sigma_2\}$'. Since $\mathfrak{Dhal}$ is not dynamically evaluated, the choice between the *then* and *else* branch cannot be made. Therefore, the set $\mathcal{S}^\vee$ denotes the set of access paths on which the choice in the original program depended on. We will call $\mathcal{S}^\vee$ the *dependency set* for the statements in the branches. Thus, there is no difference between '**if** $\mathcal{S}^\vee$ **then** $\{\sigma_1\}$ **else** $\{\sigma_2\}$' and '**if** $\mathcal{S}^\vee$ **then** $\{\sigma_2\}$ **else** $\{\sigma_1\}$' since both mean "based on $\mathcal{S}^\vee$ a choice is made between σ_1 and σ_2". We still choose to use the standard *if-then-else* structure to show the correspondence between the statement in $\mathfrak{Dhal}$ and in the language that is abstracted from.

Iteration As the iteration statement, we have '**do** $\{\sigma\}$ **while** $\mathcal{S}^\vee$' where $\mathcal{S}^\vee$ is the set of access paths which are used to determine whether iteration should continue or end. There are several reasons in favor of the *do-while* variant over the *while-do* variant of iteration. One is that the structure of a *do-while* resembles the iteration construct proposed by Böhm and Jacopini [BJ66] more closely than the structure of a *while-do*. However, the most important in our case is *prevention of code duplication*. This is shown in the next example:

Example 4.3.1 The following flow diagram can be easily translated into a *do-while* structure whereas conversion in a *while-do* structure results in code duplication:

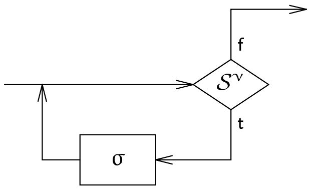


1. **do** $\{\sigma\}$ **while** $\mathcal{S}^\vee$
2. σ; **while** $\mathcal{S}^\vee$ **do** $\{\sigma\}$

The main disadvantage is that we do not know the size of σ. In the worst case σ 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 to *while-do* structures:

Example 4.3.2 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** $\mathcal{S}^\vee$ **do** $\{\sigma\}$
2. **if** $\mathcal{S}^\vee$ **then** $\{$**do** $\{\sigma\}$ **while** $\mathcal{S}^\vee\}$

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

module Dhal-struct-ctrl-syntax
imports Dhal-basic-syntax$^{4.2}$
exports
  context-free syntax
    DHALSTAT ";" DHALSTAT → DHALSTAT {**right**}
    if SET[ID] **then** "{" DHALSTAT "}" **else** "{" DHALSTAT "}" → DHALSTAT
    if SET[ID] **then** "{" DHALSTAT "}" → DHALSTAT
    do "{" DHALSTAT "}" **while** SET[ID] → DHALSTAT
    "{" DHALSTAT "}" → DHALSTAT {**bracket**}
equations

[right-assoc] $\{\sigma_1;\, \sigma_2\};\, \sigma_3 = \sigma_1;\, \sigma_2;\, \sigma_3$

[if-short] **if** $\mathcal{S}^\vee$ **then**$\{\sigma\}$ = **if** $\mathcal{S}^\vee$ **then**$\{\sigma\}$ **else**$\{\tau\}$

[rm-tau-right] $\sigma;\, \tau = \sigma$
[rm-tau-left] $\tau;\, \sigma = \sigma$

[rm-if] **if** $\mathcal{S}^\vee$ **then**$\{\sigma\}$ **else**$\{\sigma\}$ $= \sigma$

Unstructured flow of control

To denote unstructured flow of control, we have the usual '**goto**' and '**label**' statements. The semantics are as usual: 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-basic-syntax$^{4.2}$
exports
  context-free syntax
    goto ID → DHALSTAT
    label ID → DHALSTAT

4.4 Language definitions

In the next three modules, the data flow representation languages $\mathfrak{Dhal}$, s$\mathfrak{Dhal}$ (*structured* $\mathfrak{Dhal}$) and p$\mathfrak{Dhal}$ (*propagated* $\mathfrak{Dhal}$) are defined by importing the appropriate language features.

4.4.1 $\mathfrak{Dhal}$

module Dhal-syntax
imports Dhal-basic-syntax[4.2] Dhal-proc-syntax[4.2.1] Dhal-side-effect-syntax[4.3.1] Dhal-alias-syntax[4.3.1] Dhal-struct-ctrl-syntax[4.3.2] Dhal-unstruct-ctrl-syntax[4.3.2]
equations

There is a simple optimization on code with **goto** statements: if a **goto** statement is immediately followed by another **goto** statement, the second can be removed since it will never be reached.

[goto-comp] $$\textbf{goto}\ \nu_1; \textbf{goto}\ \nu_2 = \textbf{goto}\ \nu_1$$

4.4.2 $\mathfrak{sDhal}$

module Structured-dhal-syntax
imports Dhal-basic-syntax[4.2] Dhal-proc-syntax[4.2.1] Dhal-side-effect-syntax[4.3.1] Dhal-alias-syntax[4.3.1] Dhal-struct-ctrl-syntax[4.3.2]

4.4.3 $\mathfrak{pDhal}$

module Propagated-dhal-syntax
imports Dhal-basic-syntax[4.2] Dhal-proc-syntax[4.2.1] Dhal-prop-side-effect-syntax[4.3.1] Dhal-struct-ctrl-syntax[4.3.2]

Chapter 5

Control Flow Normalization

Control flow normalization (or *goto elimination*) is a transformation process on a program where unstructured cycles in the flow of control are converted (or *normalized*) into structured cycles. The major goal of control flow normalization is to simplify the syntax and structure of a program to facilitate program transformations, automatic parallelization, and program analysis such as data flow analysis. In this chapter we will give an algebraic specification of such a control flow normalization process.

5.1 Motivation

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

Second, the normalized program has a much simpler (denotational) semantics. On the one hand, a language with structured control flow can be described using so-called *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 here makes it unnecessary to perform interval analysis as part of the data flow analysis. Interval analysis is a technique to divide a (control flow) graph into sets of nodes (intervals) so that analysis of the graph can be composed from the analysis of the intervals. Normalization 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 [Amm90, Amm92].

In addition to these general advantages of control flow normalization there are considerable benefits when control flow normalization is done at the level of the data flow representation language. First, such a generic solution enables us to define a more expressive input language which makes it an easier abstraction target for the users (i.e. they can now use $\mathfrak{Dhal}$, a language with both structured and unstructured flow of control, instead of s$\mathfrak{Dhal}$, a language with only structured control flow). Second, once the choice is made to do data flow analysis on a language with only structured flow of control as s$\mathfrak{Dhal}$, generic

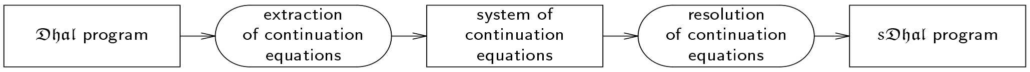


Figure 5.1: Control flow normalization of $\mathfrak{Dhal}$ programs

control flow normalization reliefs the users from implementing and performing this process themselves before or during the abstraction from source language to s$\mathfrak{Dhal}$.

5.2 Normalization Method

Normalizing the flow of control of a program P consists of transforming the program into another equivalent program P' which only uses the three control structures proposed by Böhm and Jacopini: composition, conditional and iteration [BJ66].

A lot of work has been done on control flow normalization since Böhm and Jacopini proved that it could be done for all unstructured programs and Ashcroft and Manna described the first algorithm to do so [AM72]. All these techniques convert programs with an unstructured flow of control into structured programs whose control flow graph is reducible (e.g. [WO75, Bak77, AKPW83]). A flow graph is called *reducible* iff there are no jumps into the middle of cycles from outside; consequently the only entry to a cycle is its header. A flow graph that has jumps into the middle of cycles from outside is called irreducible.

The control flow normalization technique that is used in this thesis was developed by Ammarguellat [Amm90]. It generates programs whose control flow graphs satisfy a condition that is stronger than reducibility and therefore the normalized programs are simpler in structure and syntax [Amm92]. Moreover, while some of the previous techniques result in a lot of code duplication, this technique only requires code duplication to solve irreducible flow graphs, a condition which occurs not very often, not even in unstructured programs [Hec77].

The normalization method works by transforming a program with unstructured flow of control into a system of recursive equations. The recursion variables (or unknowns) in this system represent the *continuations* associated with labels in the input program. This system of equations is solved using a Gaussian elimination-like resolution method where some transformations on the equations are done along the way. The equation that remains after resolution contains the normalized form of the program (see Figure 5.1).

The result of this transformation and resolution is that all cycles in the program are detected and all pathological syntactic constructions are eliminated (i.e. all constructions that are responsible for an unstructured flow of control). This elimination is proved to be necessary and sufficient to obtain a structured form in [WO75].

5.3 Continuations

The theory of *continuations* was first discovered by A. van Wijngaarden and independently rediscovered and developed further by F.L. Morris and C.P. Wadsworth. A good overview on the origins of continuations and related concepts is given by Reynolds [Rey93].

Definition 5.1 (Continuations) The continuation at a given point p in a program is a function which, when applied to a certain state of memory i, results in a state of memory o which is the same state as is reached when the execution of the program begins at point p using initial state i. □

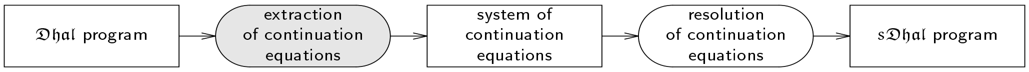


Figure 5.2: Extraction of continuation equations during control flow normalization

That is, the continuation at a given point in a program is a function which represents the rest of the program that can be reached from that point. Usually, continuations are associated with each point in a program. In the control flow normalization technique used in this thesis, such a point will be defined by a program label [Amm92].

5.3.1 Continuation Equations

Using *continuation equations*, we give a syntactic representation to the continuation at a certain program point identified by **label** ν. A continuation equation is a relation between an identifier (the recursion variable or unknown) and the part of the program which can be reached from the associated label.

module CE-syntax
imports Dhal-syntax[4.4.1]
exports
  sorts ConEq ConEqs
  context-free syntax
    Id "≡" "{" DhalStat "}" → ConEq
    ConEq∗ → ConEqs
  variables
    ε [0 − 9 ′]∗ → ConEq
    "ε*"[0 − 9 ′]∗ → ConEq∗

5.4 Finding continuations in 𝔇𝔥𝔞𝔩 programs

One of the questions that is not addressed by Ammarguellat ([Amm90, Amm92]) is how to *extract* the continuation equations from a program (see Figure 5.2). In this section we will describe a method to find these equations and prove it to be complete. The method consists of three consecutive steps (see Figure 5.3). First of all, we perform a preprocessing phase on the code which makes it easier to define the continuations. During this phase we make all control flow explicit while some of it might be implicit in the original program. Then, all program labels are converted to (nested) continuation definitions. Finally, we extract the continuation equations from this nesting starting with the innermost, replacing an extracted definition with a call to its equation along the way.

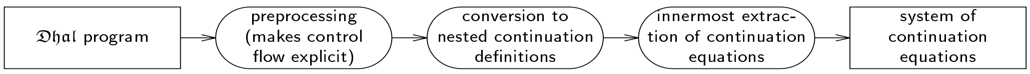


Figure 5.3: Extracting continuation equations from 𝔇𝔥𝔞𝔩 programs

5.4.1 Preprocessing

The preprocessing of $\mathfrak{Dhal}$ programs makes it easier to define the continuation equations of a program. This phase is needed since there are some cases in which it would be very difficult to define the continuations otherwise. Consider:

1. the continuation at a label in the body of a *then* or *else* branch of an *if* statement.

2. the continuation at a label in the body of a *while* statement.

In order to define the continuations for this kind of labels, we need to have more information *inside* the bodies concerning the surrounding level, since we need to know where to continue at the end of a body. Such information is added by making all control flow explicit, i.e. we transform these statements into *if* statements with additional begin- and end labels and add unstructured jumps to these new labels at the end of the bodies. Consequently, at the end of a body, there is now a reference to the continuation of that body.

The following predicate is used to determine whether a certain (series of) statements needs to be transformed:

Definition 5.2 (Basic form) We say that a (series of) $\mathfrak{Dhal}$ statement(s) σ is in *basic form* (notation $bf(\sigma) = \mathtt{true}$) when $\neg\exists\sigma_i \in \sigma.(\sigma_i \equiv \textbf{goto}\ \nu) \vee (\sigma_i \equiv \textbf{label}\ \nu)$ (i.e. the series does not contain *goto* or *label* statements). $\square$

Corollary 5.1 Statements in basic form can be expressed using $\mathsf{s}\mathfrak{Dhal}$ syntax. $\square$

The preprocessing phase $\wp$ involves the following transformation steps:

- Convert all label identifiers that were originally defined in the program to a specific lexical form in order to prevent duplicates between the original and generated labels:

$$\textbf{label}\ i \xrightarrow{\wp} \textbf{label}\ U_i$$

  We will refer to these labels as *user labels*. All *generated* labels will have identifiers with the lexical form G_i.

- Add a unique start label, say **label** G_{start}, in front of the first statement of the program or procedure. This label represents the source continuation of the program and its associated unknown will contain the normalized program after the resolution of the system.

- Make the flow of control explicit in *if* statements that are not in basic form by appending a unique generated end label and adding gotos to the new label in the branches:

$$\begin{array}{l} \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{ \\ \quad \sigma_1 \\ \}\ \textbf{else}\ \{ \\ \quad \sigma_2 \\ \} \end{array} \quad \xrightarrow{\wp} \quad \begin{array}{l} \textbf{if}\ \mathcal{S}^\nu\ \textbf{then} \\ \quad \sigma_1; \\ \quad \textbf{goto}\ G_{if_i} \\ \}\ \textbf{else}\ \{ \\ \quad \sigma_2; \\ \quad \textbf{goto}\ G_{if_i} \\ \}; \\ \textbf{label}\ G_{if_i} \end{array}$$

- Make the flow of control explicit in *while* statements that are not in basic form by translating them to *if* structures surrounded by unique generated begin- and end labels with the same operational behavior:

$$\begin{array}{lcl} & & \textbf{label } \mathsf{G}_{wh_i^b}; \\ & & \sigma; \\ \textbf{do } \{ & & \textbf{if } \mathcal{S}^{\nu} \textbf{ then} \\ \quad \sigma & & \quad \textbf{goto } \mathsf{G}_{wh_i^b} \\ \} \textbf{ while } \mathcal{S}^{\nu} & \stackrel{\wp}{\rightarrow} & \} \textbf{ else } \{ \\ & & \quad \textbf{goto } \mathsf{G}_{wh_i^e} \\ & & \}; \\ & & \textbf{label } \mathsf{G}_{wh_i^e} \end{array}$$

Possible optimizations on the transformation rules above are:

- When there is already a label defined after an *if* statement reuse that label instead of adding a new one.
- When there are already labels defined before and/or after a *while* statement reuse those labels instead of adding new ones.
- When the last statement in the body of a control structure is already a jump statement, do not add a new jump and do not generate a label if that is not necessary for the other block as well.

These optimizations are not specified because they increase the number of cases to be handled and therefore obscure the specification. Not performing these optimizations causes no real problems because during the resolution of the system these labels will be eliminated anyway. The only drawback is that the number of steps during resolution increases because trivial equations of the form $\mathsf{L}_i \equiv \{\textbf{goto } \mathsf{L}_j\}$ can be generated.

module CE-preprocess
imports Dhal-syntax$^{4.4.1}$ Dhal-tools$^{C.5}$ Naturals$^{C.2}$
exports
  context-free syntax
    $\wp$ "(" DHALPROGRAM ")" → DHALPROGRAM
hiddens
  sorts STENV DTENV
  context-free syntax
    u_{lab} "(" ID ")" → ID
    g_{lab} "(" NAT ")" → ID
The auxiliary functions for the preprocessing phase operate on so-called *environments*. These environments have two elements: the first denotes the code to be processed, the second the value of the counter which will be used for the next label to be generated.
  context-free syntax
    "(" DHALDECLS "," NAT ")" → DTENV
    "(" DHALSTAT "," NAT ")" → STENV
    $\wp_{decl}$ DTENV → DTENV
    $\wp_{stat}$ STENV → STENV
  variables
    "c$^+$" → CHAR+

equations

The conversion of user labels to their new lexical form and the generation of new labels:

[user-label] $$u_{lab}(\text{id}(c^+)) = \text{id}(\texttt{"U"}\ \texttt{"\_"}\ c^+)$$

[generate-label] $$g_{lab}(\text{nat-con}(c^+)) = \text{id}(\texttt{"G"}\ \texttt{"\_"}\ c^+)$$

The preprocessing of $\mathfrak{Dhal}$ programs is divided in preprocessing the declarations part and preprocessing the statements in the body.

[prep-prog] $$\frac{\wp_{decl}\ (\delta_1^*, 0) = (\delta_2^*, n),\ \ \wp_{stat}\ (\sigma_1, n+1) = (\sigma_2, n')}{\wp(\textbf{program}\ v\{\delta_1^*\ \sigma_1\})\ =\ \textbf{program}\ v\{\delta_2^*\ \textbf{label}\ g_{lab}(n);\ \sigma_2\}}$$

[prep-decl-empty] $$\wp_{decl}\ (, n) = (, n)$$

[prep-decl-proc] $$\frac{\wp_{stat}\ (\sigma_1, n_1+1) = (\sigma_2, n_2),\ \ \wp_{decl}\ (\delta_1^*, n_2) = (\delta_2^*, n_3)}{\wp_{decl}\ (\textbf{proc}\ v\ \mathcal{L}\{\sigma_1\}\ \delta_1^*, n_1)\ =\ (\textbf{proc}\ v\ \mathcal{L}\{\textbf{label}\ g_{lab}(n_2);\ \sigma_2\}\ \delta_2^*, n_3)}$$

[prep-decl-other] $$\frac{\wp_{decl}\ (\delta_1^*, n_1) = (\delta_2^*, n_2)}{\wp_{decl}\ (\delta\ \delta_1^*, n_1)\ =\ (\delta\ \delta_2^*, n_2)}\quad \textbf{otherwise}$$

Proposition 5.2 The function $\wp_{stat}$ is a complete function on $\mathfrak{Dhal}$ which performs the described preprocessing phase on a series of $\mathfrak{Dhal}$ statements. □

Proof Using induction on the number of statements k. Let $k = 1$. In the following rules, all possible single statements are handled using structural induction and a generic rule is given which will be applied for the cases that are not described explicitly.

[prep-label] $$\wp_{stat}\ (\textbf{label}\ v, n) = (\textbf{label}\ u_{lab}(v), n)$$

[prep-goto] $$\wp_{stat}\ (\textbf{goto}\ v, n) = (\textbf{goto}\ u_{lab}(v), n)$$

[prep-basic-if] $$\frac{\begin{array}{c} bf(\textbf{if}\ \mathcal{S}^v\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\}) = \text{true}, \\ \wp_{stat}\ (\sigma_1, n) = (\sigma_1', n_1),\ \ \wp_{stat}\ (\sigma_2, n_1) = (\sigma_2', n_2) \end{array}}{\begin{array}{l} \wp_{stat}\ (\textbf{if}\ \mathcal{S}^v\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\}, n)\ = \\ (\textbf{if}\ \mathcal{S}^v\ \textbf{then}\{\sigma_1'\}\ \textbf{else}\{\sigma_2'\}, n_2) \end{array}}$$

[prep-ext-if] $$\frac{\begin{array}{c} bf(\textbf{if}\ \mathcal{S}^v\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\}) = \text{false}, \\ \wp_{stat}\ (\sigma_1, n) = (\sigma_1', n_1),\ \ \wp_{stat}\ (\sigma_2, n_1) = (\sigma_2', n_2),\ \ g_{lab}(n_2) = v \end{array}}{\begin{array}{l} \wp_{stat}\ (\textbf{if}\ \mathcal{S}^v\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\}, n)\ = \\ (\textbf{if}\ \mathcal{S}^v\ \textbf{then}\{\sigma_1';\ \textbf{goto}\ v\}\ \textbf{else}\{\sigma_2';\ \textbf{goto}\ v\};\ \textbf{label}\ v, n_2+1) \end{array}}$$

[prep-basic-while] $$\frac{\begin{array}{c} bf(\textbf{do}\{\sigma\}\ \textbf{while}\ \mathcal{S}^v) = \text{true}, \\ \wp_{stat}\ (\sigma, n) = (\sigma', n_1) \end{array}}{\begin{array}{l} \wp_{stat}\ (\textbf{do}\{\sigma\}\ \textbf{while}\ \mathcal{S}^v, n)\ = \\ (\textbf{do}\{\sigma'\}\ \textbf{while}\ \mathcal{S}^v, n_1) \end{array}}$$

$$\text{[prep-ext-while]} \quad \frac{\begin{array}{c} bf(\mathbf{do}\{\sigma\}\ \mathbf{while}\ \mathcal{S}^{\nu}) = \text{false}, \\ \wp_{stat}\ (\sigma, n) = (\sigma', n_1),\ \ g_{lab}(n_1) = \nu_1,\ \ g_{lab}(n_1+1) = \nu_2 \end{array}}{\begin{array}{l} \wp_{stat}\ (\mathbf{do}\{\sigma\}\ \mathbf{while}\ \mathcal{S}^{\nu}, n)\ = \\ (\mathbf{label}\ \nu_1;\ \sigma';\ \mathbf{if}\ \mathcal{S}^{\nu}\ \mathbf{then}\{\mathbf{goto}\ \nu_1\}\ \mathbf{else}\{\mathbf{goto}\ \nu_2\};\ \mathbf{label}\ \nu_2,\ n_1+2) \end{array}}$$

$$\text{[prep-other]} \qquad \wp_{stat}\ (\sigma, n) = (\sigma, n) \quad \mathbf{otherwise}$$

Let $k = k_0 + 1$. The induction step is formed by the sequential composition of the series of statements with $k = k_0$ (which is correct by the induction hypothesis) and the series of statements with $k = 1$ described above:

$$\text{[prep-composition]} \qquad \frac{\begin{array}{c} \wp_{stat}\ (\sigma_1, n) = (\sigma_1', n'), \\ \wp_{stat}\ (\sigma_2, n') = (\sigma_2', n'') \end{array}}{\wp_{stat}\ (\sigma_1; \sigma_2, n)\ =\ (\sigma_1'; \sigma_2', n'')}$$

□

5.4.2 Extracting continuations

Finding the continuation equations for a series of $\mathfrak{Dhal}$ statements is done in two steps. First, the preprocessed statements are converted into a series of nested continuation definitions (dc). For this reason, the auxiliary $\mathfrak{Dhal}$ control structure **con-def** is introduced. It has two arguments: a continuation identifier and a series of statements representing the program part which corresponds to that continuation. This statement is hidden so it can only occur in the context of this module. During the conversion, each **label** L_i in the original program starts the definition of the corresponding continuation and each **goto** L_i is considered a call of the equation corresponding to L_i. Second, the continuations are extracted from the nested definitions starting with the innermost and replacing all extracted continuations with calls to their corresponding equations (f_ε).

Example 5.4.1

$$\begin{array}{ll}
& \mathbf{label}\ L_s;\ \sigma_1;\ \mathbf{if}\ \mathcal{S}^{\nu}\ \mathbf{then}\ \{\mathbf{goto}\ L_s\}\ \mathbf{else}\ \{\sigma_2;\ \mathbf{goto}\ L_i\};\ \mathbf{label}\ L_i;\ \sigma_3 \\
\overset{dc}{\twoheadrightarrow} & \mathbf{con\text{-}def(}L_s, \sigma_1;\ \mathbf{if}\ \mathcal{S}^{\nu}\ \mathbf{then}\ \{\mathbf{goto}\ L_s\}\ \mathbf{else}\ \{\sigma_2;\ \mathbf{goto}\ L_i\};\ \mathbf{con\text{-}def(}L_i, \sigma_3\mathbf{))} \\
\overset{f_\varepsilon}{\rightarrow} & \mathbf{con\text{-}def(}L_s, \sigma_1;\ \mathbf{if}\ \mathcal{S}^{\nu}\ \mathbf{then}\ \{\mathbf{goto}\ L_s\}\ \mathbf{else}\ \{\sigma_2;\ \mathbf{goto}\ L_i\};\ \mathbf{goto}\ L_i\mathbf{)} \quad \text{and} \\
& L_i \ \ \equiv\ \ \{\ \sigma_3\ \} \\
\overset{f_\varepsilon}{\rightarrow} & L_s \ \ \equiv\ \ \{\ \sigma_1;\ \mathbf{if}\ \mathcal{S}^{\nu}\ \mathbf{then}\ \{\mathbf{goto}\ L_s\}\ \mathbf{else}\ \{\sigma_2;\ \mathbf{goto}\ L_i\}\ ^{\dagger}\ \} \\
& L_i \ \ \equiv\ \ \{\ \sigma_3\ \}
\end{array}$$

† The extraction for *if* statements that are not in basic form is a special case in which the trailing call can be omitted (see also equation [extr-ce-ext-if] on the following page). □

module CE-find
imports Dhal-syntax[4.4.1] Dhal-tools[C.5] CE-syntax[5.3.1]
exports
  context-free syntax
    f_ε DHALSTAT → CONEQS

hiddens

sorts CEENV

context-free syntax

dc "(" DHALSTAT ")" → DHALSTAT
con-def "(" ID "," DHALSTAT ")" → DHALSTAT
"(" DHALSTAT "," CONEQS ")" → CEENV
e_ε CEENV → CEENV

equations

The function that finds the continuation equations of a series of statements defines the continuations and extracts them using auxiliary functions:

[find-ce]
$$\frac{e_\varepsilon\ (dc(\sigma),\ ) = (\sigma',\ \varepsilon^*)}{f_\varepsilon\ \sigma\ =\ \varepsilon^*}$$

Proposition 5.3 The function *dc* is a complete function on $\mathfrak{Dhal}$ statements which converts a series of $\mathfrak{Dhal}$ statements into nested continuation definitions. □

Proof Using induction on the number of statements k. The basic step for k = 1 proves that *dc* defines the continuations for a single statement using structural induction:

[def-con-single-label] $dc(\textbf{label}\ \nu) = \textbf{con-def}(\nu,\ \tau)$

[def-con-single-if] $dc(\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\}) = \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{dc(\sigma_1)\}\ \textbf{else}\{dc(\sigma_2)\}$

[def-con-single-while] $dc(\textbf{do}\{\sigma_1\}\ \textbf{while}\ \mathcal{S}^\nu) = \textbf{do}\{dc(\sigma_1)\}\ \textbf{while}\ \mathcal{S}^\nu$

The following equation uses the predicate $1(\sigma)$ on $\mathfrak{Dhal}$ statements which determines wether a statement is a composition of other statements (then $1(\sigma)$ = false) or not (then $1(\sigma)$ = true). This predicate is defined in the module Dhal-tools[C.5].

[def-con-single-other] $dc(\sigma) = \sigma$ **when** $1(\sigma) = \text{true}$

The induction step (k > 1) uses structural induction on the first statement of the series. A program label is a point for which a continuation has to be defined. By definition this is the tail of the series (and implicitly everything that can be reached from the tail using jumps).

[def-con-label] $dc(\textbf{label}\ \nu;\ \sigma) = \textbf{con-def}(\nu,\ dc(\sigma))$

[def-con-comp] $dc(\sigma_1;\ \sigma_2) = dc(\sigma_1);\ dc(\sigma_2)$ **otherwise**

□

Proposition 5.4 The function e_ε is a complete function on $\mathfrak{Dhal}$ which extracts all continuation equations from a series of nested continuation definitions. It starts with the innermost definition and replaces each extracted continuation with a call to its corresponding equation. □

Proof Using induction on the number of statements k. The basic step for $k = 1$ uses structural induction on single statements:

[extr-ce-con-def] $$\frac{e_\varepsilon\ (\sigma_1,\ \varepsilon_1^*) = (\sigma_1',\ \varepsilon_2^*)}{e_\varepsilon\ (\textbf{con-def}(\nu,\ \sigma_1),\ \varepsilon_1^*)\ =\ (\textbf{goto}\ \nu,\ \nu \equiv \{\sigma_1'\}\ \varepsilon_2^*)}$$

[extr-ce-if] $$\frac{e_\varepsilon\ (\sigma_1,\ ) = (\sigma_1',\ \varepsilon_1^*),\ \ e_\varepsilon\ (\sigma_2,\ ) = (\sigma_2',\ \varepsilon_2^*)}{\begin{array}{c} e_\varepsilon\ (\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\},\ \varepsilon^*)\ = \\ (\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_1'\}\ \textbf{else}\{\sigma_2'\},\ \varepsilon^*\ \varepsilon_1^*\ \varepsilon_2^*)\end{array}}$$

[extr-ce-while] $$\frac{e_\varepsilon\ (\sigma_1,\ ) = (\sigma_1',\ \varepsilon_1^*)}{e_\varepsilon\ (\textbf{do}\{\sigma_1\}\ \textbf{while}\ \mathcal{S}^\nu,\ \varepsilon^*)\ =\ (\textbf{do}\{\sigma_1'\}\ \textbf{while}\ \mathcal{S}^\nu,\ \varepsilon^*\ \varepsilon_1^*)}$$

[extr-ce-other] $$e_\varepsilon\ (\sigma,\ \varepsilon^*) = (\sigma,\ \varepsilon^*)\quad \textbf{when}\ \ \mathit{1}(\sigma) = \text{true}\quad \textbf{otherwise}$$

The induction step for $k > 1$ uses structural induction on the first argument:

Suppose the first statement is an *if* statement that is not in basic form. After preprocessing, such a statement will always be followed by an end label, and consequently, a continuation definition. Furthermore, the branches of such a statement have calls to the continuation equation corresponding to this end label as their final statements. For this reason, we do not have to generate a continuation call following the *if* statement. However, we do need to process the statements following the *if* statement in order to find the equations:

[extr-ce-ext-if] $$\frac{\begin{array}{c} \mathit{bf}(\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\}) = \text{false}, \\ e_\varepsilon\ (\sigma_1,\ ) = (\sigma_1',\ \varepsilon_1^*),\ \ e_\varepsilon\ (\sigma_2,\ ) = (\sigma_2',\ \varepsilon_2^*),\ \ e_\varepsilon\ (\sigma_3,\ ) = (\sigma_3',\ \varepsilon_3^*)\end{array}}{\begin{array}{c} e_\varepsilon\ (\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\};\ \sigma_3,\ \varepsilon^*)\ = \\ (\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_1'\}\ \textbf{else}\{\sigma_2'\},\ \varepsilon^*\ \varepsilon_1^*\ \varepsilon_2^*\ \varepsilon_3^*)\end{array}}$$

In all other cases (the first statement is not an *if* statement or it is an *if* statement in basic form), we process both parts of the composition and combine the results.

[extr-ce-comp] $$\frac{e_\varepsilon\ (\sigma_1,\ ) = (\sigma_1',\ \varepsilon_1^*),\ \ e_\varepsilon\ (\sigma_2,\ ) = (\sigma_2',\ \varepsilon_2^*)}{e_\varepsilon\ (\sigma_1;\ \sigma_2,\ \varepsilon^*)\ =\ (\sigma_1';\ \sigma_2',\ \varepsilon^*\ \varepsilon_1^*\ \varepsilon_2^*)}\quad \textbf{otherwise}$$

□

From this point on, we will omit proofs for functions in the specifications like the ones above and in the previous module. However, the remaining specifications are presented using the same structure allowing the reader to visualize the accompanying proofs.

Example 5.4.2 This is an example of the step-wise extraction of continuation equations from a 𝔇𝔥𝔞𝔩 program. Consider the following input program, the definition of its continuations and the system of continuation equations after extraction:

input program		continuation definitions
label $\mathsf{L_s}$;		**con-def**($\mathsf{L_s}$,
σ_1;		σ_1;
if $\mathcal{S}^\vee$ **then** {		**if** $\mathcal{S}^\vee$ **then** {
label L;		**con-def**(L,
σ_2;		σ_2;
goto $\mathsf{L_i}$		**goto** $\mathsf{L_i}$)
} **else** {	$\overset{dc}{\twoheadrightarrow}$	} **else** {
σ_3;		σ_3;
goto $\mathsf{L_i}$		**goto** $\mathsf{L_i}$
};		};
label $\mathsf{L_i}$;		**con-def**($\mathsf{L_i}$,
σ_4		σ_4))

$\downarrow e_\varepsilon$

continuation equations
$\mathsf{L_s} \equiv \{\sigma_1;$**if** $\mathcal{S}^\vee$ **then** {**goto** L} **else** $\{\sigma_3;$**goto** $\mathsf{L_i}\}\}$
$\mathsf{L} \equiv \{\sigma_2;$**goto** $\mathsf{L_i}\}$
$\mathsf{L_i} \equiv \{\sigma_4\}$

□

5.4.3 Remarks

During the extraction of the continuation equations using the algorithm described above, the structure of the original program is preserved as much as possible. If the input program contains control structures that are already in basic form, these constructs are preserved and not transformed for re-normalization. The continuation equations that are found contain no unstructured flow of control and have a highly regular structure:

$$
\begin{array}{lcl}
\textit{equation} & \rightarrow & \mathsf{L_i} \equiv \{\textit{body}\} \\
\textit{body} & \rightarrow & \sigma;\ \textbf{goto}\ \mathsf{L_j} \\
 & | & \sigma_1;\ \textbf{if}\ \mathcal{S}^\vee\ \textbf{then}\ \{\textit{body}\}\ \textbf{else}\ \{\textit{body}\}
\end{array}
$$

where σ_n is a series of 𝔇𝔥𝔞𝔩 statements in basic form. Note that the equations still use the *goto* statement but that it is given a slightly different semantics here: the statements **goto** $\mathsf{L_i}$ will be used to denote a call of the equation identified by $\mathsf{L_i}$. We have made the choice to reuse the *goto* statement for this purpose because it meets the intuition and prevents us from having to extend the 𝔇𝔥𝔞𝔩 syntax.

5.5 Transformations on continuation equations

The resolution of the system of continuation equations is based on Gaussian elimination. This consists of the repeated substitution and elimination of the unknowns in a system of equations. We want the resolution to satisfy two requirements:

1. to minimize the code size
2. to convert every control flow cycle into a *while* loop

In order to meet these requirements, two transformations (factorization and derecursivation) are performed on the equations during the resolution process. These transformations will be presented in the remainder of this section, but first, we define two additional predicates on continuation equations:

Definition 5.3 (Call set) The call set *calls* of a continuation equation ε is the set of unknowns ν_i that are called in the body of ε: $calls_{\nu\equiv\{\sigma\}} \Leftrightarrow \{\nu_i |\ \textbf{goto}\ \nu_i \in \sigma\}$. □

Definition 5.4 (Self recursive) A continuation equation ε is self recursive ($\circlearrowleft$) when it calls itself: $\circlearrowleft_{\nu\equiv\{\sigma\}} \Leftrightarrow \nu \in calls_{\nu\equiv\{\sigma\}}$. □

5.5.1 Factorization

Factorization (F) is the transformation that is used to keep the code size minimal. An unknown ν may be called several times in a continuation equation ε. To avoid code duplication when substituting the body of ν in ε, we first *factor* the occurrences of ν in ε. In order to do this, we have to assemble the *continuation dependency set* of ν in ε.

Definition 5.5 (Continuation dependency set) The *continuation dependency set* of an unknown ν in an equation ε is the union of the dependency sets guarding all calls of ν in ε. That is, it is the union of the guards of all conditional constructs in which ν was called. □

Example 5.5.1 Consider the following continuation equation:

$$L_1 \equiv \{\sigma_1; \textbf{if}\ \mathcal{S}_1^\nu\ \textbf{then}\ \{\textbf{goto}\ L_2\}\ \textbf{else}\ \{\textbf{if}\ \mathcal{S}_2^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\textbf{goto}\ L_3\}\}\}$$

In this equation, the continuation dependency set for L_2 is $\mathcal{S}_1^\nu$ and the continuation dependency set for L_3 is $\mathcal{S}_1^\nu \cup \mathcal{S}_2^\nu$. □

Factorization of an unknown ν in a continuation equation ε consists of assembling the continuation dependency set of ν in ε and replacing the body of the equation by a new body in which ν is called only once, guarded by this continuation dependency set.

The union is sufficient here since conditional constructs in $\mathfrak{Dhal}$ only have static semantics and their guards are sets of variables. To process a language that has conditionals with the usual dynamic semantics, one should build (boolean) expressions in which the branch where the call occurs is taken into account, i.e. the else branch depends on the negation of the condition.

5.5.2 Derecursivation

Derecursivation (D) is the transformation that is used to convert explicit control flow cycles into *while* loops. An explicit cycle in the flow of control will, at some time during the resolution, result in a self recursive continuation equation. The derecursivation transformation converts such a self recursive equation into a *while* loop.

Example 5.5.2 Let $L_i \equiv \{\ \sigma_1;\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\textbf{goto}\ L_i\}\ \textbf{else}\ \{\textbf{goto}\ L_j\}\ \}$ be a self-recursive equation. Obviously, this equation represents a loop whose iteration depends on $\mathcal{S}^\nu$. Derecursivation transforms this equation into $L_i \equiv \{\ \textbf{do}\ \{\sigma_1\}\ \textbf{while}\ \mathcal{S}^\nu;\ \textbf{goto}\ L_j\ \}$ □

Similar to the factorization transformation, we need to assembly the continuation dependency set concerning the self-recursive unknown in order to generate the *while* loop since a recursive call may occur at any nesting levels.

5.5.3 The specification

module CE-trans
imports CE-syntax[5.3.1] Dhal-tools[C.5]
exports
  context-free syntax
    F "(" CONEQ ")" $\rightarrow$ CONEQ
    D "(" CONEQ ")" $\rightarrow$ CONEQ
    S "(" CONEQ "," CONEQ ")" $\rightarrow$ CONEQ
    $\circlearrowleft$ "_" CONEQ $\rightarrow$ BOOL
    calls "_" CONEQ $\rightarrow$ SET[ID]
hiddens
  sorts DENV
  context-free syntax
    S "(" CONEQ "," DHALSTAT ")" $\rightarrow$ DHALSTAT
    calls "_" DHALSTAT $\rightarrow$ SET[ID]
    2_{if} SET[ID×SET[ID]] $\rightarrow$ DHALSTAT
    "(" DHALSTAT "," DHALSTAT "," SET[ID×SET[ID]] ")" $\rightarrow$ DENV
    dis "(" DHALSTAT "," SET[ID] "," SET[ID×SET[ID]] ")" $\rightarrow$ DENV
equations

The predicate *calls* denotes the set of unknowns that are called in a continuation equation ε. As described before, the statement **goto** L_i is considered a call of the equation identified by L_i.

[calls-coneq] $$calls_{\nu\ \equiv\{\sigma\}} = calls_{\sigma}$$

[calls-comp] $$calls_{\sigma_1;\ \sigma_2} = calls_{\sigma_1} \cup calls_{\sigma_2}$$

[calls-goto] $$calls_{\mathbf{goto}\ \nu} = \{\nu\}$$

[calls-if] $$calls_{\mathbf{if}\ \mathcal{S}^{\nu}\ \mathbf{then}\{\sigma_1\}\ \mathbf{else}\{\sigma_2\}} = calls_{\sigma_1} \cup calls_{\sigma_2}$$

[calls-while] $$calls_{\mathbf{do}\{\sigma\}\ \mathbf{while}\ \mathcal{S}^{\nu}} = calls_{\sigma}$$

[calls-other] $$calls_{\sigma} = \{\} \quad \textbf{otherwise}$$

An equation is self recursive when it calls itself

[self-recursive] $$\circlearrowleft_{\nu\ \equiv\{\sigma\}} = \nu \in calls_{\nu\ \equiv\{\sigma\}}$$

The transformations F and D depend on structural analysis and dissection of the code under transformation. For this purpose a generic auxiliary function *dis* is used that analyzes a series of statements and dissects it into two versions:

1. a copy where all continuation calls are removed. We will call this the *continuation-less* version.

2. a copy where only continuation calls and the control flow structure that is necessary for those calls is left over. We will call this the *continuation-only* version.

Furthermore, *dis* assembles a set of tuples $(\nu_i, \mathcal{S}_i^{\nu})$ denoting the dependency set $\mathcal{S}_i^{\nu}$ for every continuation ν_i that is called in the input. The *dis* function operates on a three-tuple: (DHALSTAT × SET[ID] × SET[ID×SET[ID]]), describing the input series of statements, the set of variables on which this series depends (for analysis of compound statements like *if* and *while*) and the set of continuation dependencies found so far and it generates a three-tuple: (DHALSTAT × DHALSTAT × IDTUPSET) describing the two versions and the new set of continuation dependencies

[dissect-goto]
$$\frac{(\nu,\, \mathcal{S}^{\nu}) \hookrightarrow \mathcal{S}^{\theta} = \mathcal{S}^{\theta\,\prime}}{\mathit{dis}(\mathbf{goto}\ \nu,\, \mathcal{S}^{\nu},\, \mathcal{S}^{\theta}) \;=\; (\tau,\, \mathbf{goto}\ \nu,\, \mathcal{S}^{\theta\,\prime})}$$

[dissect-if]
$$\frac{\begin{array}{c}\mathit{dis}(\sigma_1,\, \mathcal{S}_1^{\nu} \cup \mathcal{S}_2^{\nu},\, \mathcal{S}_1^{\theta}) = (\sigma_5,\, \sigma_6,\, \mathcal{S}_2^{\theta}),\\ \mathit{dis}(\sigma_2,\, \mathcal{S}_1^{\nu} \cup \mathcal{S}_2^{\nu},\, \mathcal{S}_2^{\theta}) = (\sigma_7,\, \sigma_8,\, \mathcal{S}_3^{\theta})\end{array}}{\begin{array}{l}\mathit{dis}(\mathbf{if}\ \mathcal{S}_1^{\nu}\ \mathbf{then}\{\sigma_1\}\ \mathbf{else}\{\sigma_2\},\, \mathcal{S}_2^{\nu},\, \mathcal{S}_1^{\theta}) \;=\\ (\mathbf{if}\ \mathcal{S}_1^{\nu}\ \mathbf{then}\{\sigma_5\}\ \mathbf{else}\{\sigma_7\},\, \mathbf{if}\ \mathcal{S}_1^{\nu}\ \mathbf{then}\{\sigma_6\}\ \mathbf{else}\{\sigma_8\},\, \mathcal{S}_3^{\theta})\end{array}}$$

[dissect-other]
$$\mathit{dis}(\sigma,\, \mathcal{S}^{\nu},\, \mathcal{S}^{\theta}) = (\sigma,\, \tau,\, \mathcal{S}^{\theta}) \quad \mathbf{otherwise}$$

Note that there is no special rule for *while* loops since the only *while* loops that are possible after preprocessing are in basic form, i.e. their bodies do not contain continuation calls. Consequently, they can be handled using the default case.

[dissect-comp]
$$\frac{\begin{array}{c}\mathit{dis}(\sigma_1,\, \mathcal{S}^{\nu},\, \mathcal{S}^{\theta}) = (\sigma_3,\, \sigma_4,\, \mathcal{S}_1^{\theta}),\\ \mathit{dis}(\sigma_2,\, \mathcal{S}^{\nu},\, \mathcal{S}_1^{\theta}) = (\sigma_5,\, \sigma_6,\, \mathcal{S}_2^{\theta})\end{array}}{\mathit{dis}(\sigma_1;\, \sigma_2,\, \mathcal{S}^{\nu},\, \mathcal{S}^{\theta}) \;=\; (\sigma_3;\, \sigma_5,\, \sigma_4;\, \sigma_6,\, \mathcal{S}_2^{\theta})}$$

Factorization of an equation consists of dissecting its body into a continuation-less and a continuation-only version and concatenating these statements in a new body.

[factor-coneq]
$$\frac{\mathit{dis}(\sigma,\, \{\},\, \{\}) = (\sigma_1,\, \sigma_2,\, \mathcal{S}^{\theta}),\;\; \mathcal{Z}_{\mathit{if}}\ \mathcal{S}^{\theta} = \sigma_3}{F(\nu \equiv \{\sigma\}) \;=\; \nu \equiv \{\sigma_1;\, \sigma_3\}}$$

Derecursivation of an equation needs only to be done when the equation is self recursive. When an equation is self recursive, it can be derecursivated by replacing its body by the concatenation of (a) the continuation-less version surrounded by a *while* loop and (b) the continuation-only version from which the recursive call(s) are removed.

[derec-non-selfrec-coneq]
$$\frac{\circlearrowleft_{\nu\, \equiv \{\sigma\}} = \text{false}}{D(\nu \equiv \{\sigma\}) \;=\; \nu \equiv \{\sigma\}}$$

[derec-selfrec-coneq]
$$\frac{\begin{array}{c}\circlearrowleft_{\nu\, \equiv \{\sigma\}} = \text{true},\\ \mathit{dis}(\sigma,\, \{\},\, \{\}) = (\sigma_1,\, \sigma_2,\, \{\theta_1^*,\, (\nu,\, \mathcal{S}^{\nu}),\, \theta_2^*\}),\\ \tau_{\mathbf{goto}\ \nu}(\sigma_2) = \sigma_3\end{array}}{D(\nu \equiv \{\sigma\}) \;=\; \nu \equiv \{\mathbf{do}\{\sigma_1\}\ \mathbf{while}\ \mathcal{S}^{\nu};\, \sigma_3\}}$$

The substitution function (S) substitutes all calls of a given equation with the body of that equation. It distributes over *if* statements and composition and is the identity on all other statements. Note that *while* statements can be handled using the default case since they can not contain continuation calls after the preprocessing phase.

[subst-coneq]
$$S(\varepsilon,\, \nu \equiv \{\sigma\}) = \nu \equiv \{S(\varepsilon,\, \sigma)\}$$

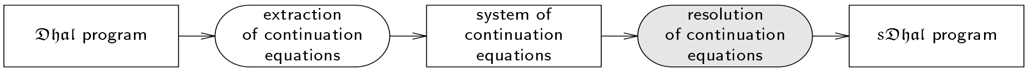


Figure 5.4: Resolution during control flow normalization

[subst-goto] $$S(\nu \equiv \{\sigma\}, \textbf{goto}\ \nu) = \sigma$$

[subst-if] $$\frac{S(\nu \equiv \{\sigma\}, \sigma_1) = \sigma_3,\ \ S(\nu \equiv \{\sigma\}, \sigma_2) = \sigma_4}{S(\nu \equiv \{\sigma\}, \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\})\ =\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_3\}\ \textbf{else}\{\sigma_4\}}$$

[subst-other] $$S(\nu \equiv \{\sigma\}, \sigma_1) = \sigma_1 \quad \textbf{otherwise}$$

[subst-comp] $$\frac{S(\nu \equiv \{\sigma\}, \sigma_1) = \sigma_3,\ \ S(\nu \equiv \{\sigma\}, \sigma_2) = \sigma_4}{S(\nu \equiv \{\sigma\}, \sigma_1; \sigma_2)\ =\ \sigma_3; \sigma_4}$$

The function $\mathcal{2}_{\mathit{if}}$ converts a set of continuation dependencies to a sequence of *if* statements, one for every continuation, with the corresponding dependency set as "condition".

[to-if-empty] $$\mathcal{2}_{\mathit{if}}\ \{\} = \tau$$

[to-if-recurse] $$\frac{\mathcal{2}_{\mathit{if}}\ \{\theta^*\} = \sigma}{\mathcal{2}_{\mathit{if}}\ \{(\nu, \mathcal{S}^\nu), \theta^*\}\ =\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\textbf{goto}\ \nu\}; \sigma}$$

5.6 Resolution

The technique to solve a system of continuation equations is based on Gaussian elimination-like resolution. This resolution is the final step during control flow normalization (see Figure 5.4). It consists of repeatedly performing the following steps using some (predetermined) order $\mathcal{O}$ on the unknowns of a system $\mathfrak{S}$:

1. Choose the first unknown ν and eliminate its defining equation ε_ν from the system
2. Derecursivate ε_ν if necessary
3. Check all remaining equations ε_i in the system:
   if ε calls ν then factorize ε and substitute the body of ε_ν for the call of ν.

module Resolution
imports CE-syntax[5.3.1] CE-trans[5.5.3] Identifier-Containers[C.4]
exports
  context-free syntax
    Σ "(" LIST[ID] "," CONEQS ")" $\rightarrow$ DHALSTAT
hiddens
  context-free syntax
    S^* "(" CONEQ "," CONEQS ")" $\rightarrow$ CONEQS

equations

The function Σ processes the system of equations following some predetermined order. When the system is empty, nothing has to be done.

[solve-empty] $$\Sigma(\mathcal{L},\ ) = \tau$$

An iteration step consists of the elimination of an unknown from the system, derecursivating it and then substituting it in the remaining system.

[solve-iterate]
$$\begin{aligned} &\Sigma([\nu_1, \nu_2, \nu^*],\ \varepsilon_1^*\ \nu_2 \equiv\{\sigma\}\ \varepsilon_2^*) \\ = \ &\Sigma([\nu_1, \nu^*],\ S^*(D(\nu_2 \equiv\{\sigma\}),\ \varepsilon_1^*\ \varepsilon_2^*)) \end{aligned}$$

The resolution is finished when the final continuation equation is derecursivated.

[solve-ready]
$$\frac{D(\nu \equiv\{\sigma\}) = \nu' \equiv\{\sigma'\}}{\Sigma([\nu],\ \varepsilon_1^*\ \nu \equiv\{\sigma\}\ \varepsilon_2^*)\ =\ \sigma'}$$

The substitution of a given continuation equation in an empty system of continuations results in an empty system.

[substs-empty] $$S^*(\nu \equiv\{\sigma\},\ ) =$$

When the the first equation of the system (ε_f) calls the equation to substitute (ε_s), we factorize that first equation, substitute ε_s in the factorized version of ε_f, and continue with the substitution of ε_s in the remaining system.

[substs-first]
$$\frac{\begin{array}{c} \nu_1 \in \mathit{calls}_{\varepsilon_1} = \text{true}, \\ S(\nu_1 \equiv\{\sigma\},\ F(\varepsilon_1)) = \varepsilon_2, \\ S^*(\nu_1 \equiv\{\sigma\},\ \varepsilon_1^*) = \varepsilon_2^* \end{array}}{S^*(\nu_1 \equiv\{\sigma\},\ \varepsilon_1\ \varepsilon_1^*)\ =\ \varepsilon_2\ \varepsilon_2^*}$$

When the the first equation of the system does not call the equation to substitute (ε_s), we immediately continue with the substitution of ε_s in the remaining system.

[substs-rest]
$$\frac{\begin{array}{c} \nu_1 \in \mathit{calls}_{\varepsilon} = \text{false}, \\ S^*(\nu_1 \equiv\{\sigma\},\ \varepsilon_1^*) = \varepsilon_2^* \end{array}}{S^*(\nu_1 \equiv\{\sigma\},\ \varepsilon\ \varepsilon_1^*)\ =\ \varepsilon\ \varepsilon_2^*}$$

5.7 Order of resolution

When Gaussian elimination is used for the resolution of linear algebra systems, the order in which the unknowns are eliminated and substituted in the system is not very important for the quality of the solution since it is found by evaluating the expression that remains after resolution. If the expression increases due to wrong choices in the order, this evaluation may take a little longer but that is only a minor inconvenience. On the other hand, in the case of control flow normalization the normalized program is the expression that is found after the resolution of a system of continuation equations. If that expression increases due to wrong choices in the resolution order, it will affect all processes that use the program after control flow normalization (e.g. program transformation, automatic parallelization, and data flow analysis). Consequently, the order is very important for the quality of this solution since some orderings result in code duplication and therefore increase the size of this expression. This can be seen easily in the following example:

Example 5.7.1 Let us define the following system of continuation equations:

$$\mathfrak{S} := \begin{cases} \mathsf{L}_0 \equiv \{\ \sigma_1;\ \textbf{goto}\ \mathsf{L}_1\ \} \\ \mathsf{L}_1 \equiv \{\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2;\ \textbf{goto}\ \mathsf{L}_2\}\ \textbf{else}\ \{\sigma_3\}\ \} \\ \mathsf{L}_2 \equiv \{\ \sigma_4;\ \textbf{goto}\ \mathsf{L}_1\ \} \end{cases}$$

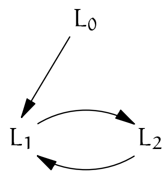


Figure 5.5: Dependency graph of $\mathfrak{S}$

with L_0 the root equation. The dependency graph of this system is shown in Figure 5.5. Since L_0 is the root equation, this is the equation that should hold the solution of the system after the resolution. Consequently, there are two possible orders for the resolution of this system:

1. The order $[\mathsf{L}_0, \mathsf{L}_1, \mathsf{L}_2]$
   (a) derecursivation of L_1 is not necessary
   (b) factorization of L_0 is not necessary
   (c) substitute L_1 in $\mathsf{L}_0 \Rightarrow \mathsf{L}_0 \equiv \{\ \sigma_1;\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2;\ \textbf{goto}\ \mathsf{L}_2\}\ \textbf{else}\ \{\sigma_3\}\ \}$
   (d) factorization of L_2 is not necessary
   (e) substitute L_1 in $\mathsf{L}_2 \Rightarrow \mathsf{L}_2 \equiv \{\ \sigma_4;\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2;\ \textbf{goto}\ \mathsf{L}_2\}\ \textbf{else}\ \{\sigma_3\}\ \}$
   (f) derecursivate $\mathsf{L}_2 \Rightarrow \mathsf{L}_2 \equiv \{\ \textbf{do}\ \{\sigma_4;\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\sigma_3\}\}\ \textbf{while}\ \mathcal{S}^\nu\ \}$
   (g) factor $\mathsf{L}_0 \Rightarrow \mathsf{L}_0 \equiv \{\ \sigma_1;\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\sigma_3\};\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\textbf{goto}\ \mathsf{L}_2\}\ \}$
   (h) substitute L_2 in L_0
   $\Rightarrow \mathsf{L}_0 \equiv \{\ \sigma_1;\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\sigma_3\};$
   $\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\textbf{do}\ \{\sigma_4;\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\sigma_3\}\}\ \textbf{while}\ \mathcal{S}^\nu\}\ \}$
2. The order $[\mathsf{L}_0, \mathsf{L}_2, \mathsf{L}_1]$
   (a) derecursivation of L_2 is not necessary
   (b) factor $\mathsf{L}_1 \Rightarrow \mathsf{L}_1 \equiv \{\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\sigma_3\};\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\textbf{goto}\ \mathsf{L}_2\}\ \}$
   (c) substitute L_2 in L_1
   $\Rightarrow \mathsf{L}_1 \equiv \{\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\sigma_3\};\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_4;\ \textbf{goto}\ \mathsf{L}_1\}\ \}$
   (d) derecursivate L_1
   $\Rightarrow \mathsf{L}_1 \equiv \{\ \textbf{do}\ \{\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\sigma_3\};\ \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_4\}\}\ \textbf{while}\ \mathcal{S}^\nu\ \}$
   (e) factorization of L_0 is not necessary
   (f) substitute L_1 in L_0
   $\Rightarrow \mathsf{L}_0 \equiv \{\ \sigma_1; \textbf{do}\ \{\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_2\}\ \textbf{else}\ \{\sigma_3\}; \textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\ \{\sigma_4\}\}\ \textbf{while}\ \mathcal{S}^\nu\ \}$

Obviously, the resulting expressions are semantically equivalent, but the second one does not contain any duplicated code while the first one does. This can seriously increase the code size, especially when σ_2 or σ_3 are big expressions. □

The problem is how to find an ordering on the unknowns of a system of continuation equations which results in minimal code duplication during resolution. Before we describe our algorithm, we make the following observations which give a rationale for our approach:

- The factorization transformation has most benefits for preventing code duplication when it is applied to a continuation equation ε with as much calls of the same unknown ν as possible. The reason is that all these calls are replaced by a single call that is guarded by their combined continuation dependency set.

- The derecursivation transformation has most benefits for preventing code duplication when it is applied to a continuation equation which does not call unknows whose equations call that continuation equation. The reason is that after substitution of these unknowns, the equation will be self-recursive again and derecursivation will lead to nested *while* loops.

One way to create the kind of continuation equations described in the first observation is to repeatedly substitute the unknowns that are called by the root equation in the root equation starting with the ones that are not called by others. In terms of dependency graphs this corresponds to repeatedly substituting the successor nodes of the root in the root. For an acyclic graph $\mathcal{G}$, such an ordering is defined by the so-called *topological sort* of the nodes of $\mathcal{G}$ [Eve79]. However, in general, our input graphs can contain cycles. The solution is to first eliminate the back- and cross edges and topologically sort the remaining graph. We call this an *extended topological sort* of the graph. The algorithm is formally specified by equations eto-ready and eto-recurse on page 45. It performs a *topological sort* and ignores the edges to nodes which are already in the ordering since these are considered back- or cross edges.

To meet the second observation, we must substitute all nodes that occur in the body of a cycle into the header of that cycle. This can be done by interchanging the positions of the cycle header and the nodes from its body in the extended topological sorting. Unfortunately, this has the disadvantage that it might disturb the ordering between unknowns that have nothing to do with the cycle, as is shown in the next example. For now, we will stick to this method to determine the resolution order although it might be not optimal in some cases. The simple reason is that we have not been able to find a better algorithm and it is not a burning issue for our data flow analysis purposes.

Example 5.7.2 Let $\mathcal{D}$ be the dependency graph of a given system of continuation equations with root x_0 and cycle $(x_1 \rightarrow x_4 \rightarrow x_5 \rightarrow x_1)$ with header x_1 and body $\{x_4, x_5\}$ (see Figure 5.6).
One of the possible orderings which can be found by an extended topological sort of the nodes of $\mathcal{D}$ is $[x_0, x_1, x_2, x_3, x_4, x_5]$. When this ordering is adapted to meet the second observation by moving the body before the header, we get the ordering $[x_0, x_4, x_5, x_1, x_2, x_3]$ which does not have the sub-ordering $x_2 \rightarrow x_4$ that was available in $\mathcal{D}$. On the other hand, when the header is moved behind the body, we get $[x_0, x_2, x_3, x_4, x_5, x_1]$ which does not have the sub-ordering $x_1 \rightarrow x_3$ that was available in the graph. Obviously, when we completely interchange the body and the header, we get both faults (the order becomes $[x_0, x_4, x_5, x_2, x_3, x_1]$ so neither $x_2 \rightarrow x_4$ nor $x_1 \rightarrow x_3$ are available).

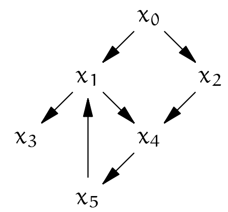


Figure 5.6: Graph $\mathcal{D}$

□

An algorithm to find all cycles in a graph and their headers is described in [RND77]. This algorithm uses several global data-types (i.e. multidimensional arrays, sets and a stack). In an ASF+SDF specification, these all have to be coded in the parameters of the functions in some way, since ASF+SDF has no global data space. Such an encoding would not increase the comprehensibility of the specification so we will not describe the algorithm using ASF+SDF. Instead, the complete algorithm is described using pseudo-code in algorithms 1, 2 and 3. In the specification which follows the algorithms, we use a function f_{loops} which acts as a dummy for the find_all_cycles procedure.

```
procedure cycle(v : node, var f : bool) :=
    flag ← false
    push(v, path)
    avail(v) ← false
    foreach w ∈ Adj(v) do
        if w = s then cycles_found ← cycles_found ∪ {(s, path)}
                        flag ← true
                  else if avail(w) then cycle(w, g)
                                            flag ← flag ∨ g
    if flag then unmark(v)
            else foreach w ∈ Adj(v) do B(w) ← B(w) ∪ {v}
    v ← pop(path)
    return
```

Algorithm 1: The procedure that performs a depth-first search for a cycle

module Resolution-order
imports CE-syntax[5.3.1] CE-trans[5.5.3] Identifier-Containers[C.4]
exports
  context-free syntax
    $\mathcal{O}$ CONEQS → LIST[ID]
hiddens
  sorts TSENV
  context-free syntax
    f_{graph} "(" SET[ID] "," SET[ID] "," SET[ID×SET[ID]] "," CONEQS ")" → SET[ID×SET[ID]]
    $\mathcal{O}_{xt}$ "(" LIST[ID] "," LIST[ID] "," SET[ID×SET[ID]] ")" → LIST[ID]
    f_{loops} "(" SET[ID×SET[ID]] ")" → SET[ID×LIST[ID]]
    $\mathcal{O}_{ch}$ "(" LIST[ID] "," SET[ID×LIST[ID]] ")" → LIST[ID]
equations

[order-empty]
$$\mathcal{O} = []$$

[order-other]
$$\frac{f_{graph}(\{\nu\},\ \{\},\ \{\},\ \nu \equiv \{\sigma\}\ \varepsilon^*) = \mathcal{S}^\theta,\quad \mathcal{O}_{ch}(\mathcal{O}_{xt}([\nu],\ [],\ \mathcal{S}^\theta),\ f_{loops}(\mathcal{S}^\theta)) = \mathcal{L}}{\mathcal{O}\ \nu \equiv \{\sigma\}\ \varepsilon^*\ =\ \mathcal{L}}$$

The f_{graph} function finds the dependency graph for a system of continuation equations reachable from a given (set of) unknowns. The graph is coded in adjacency structure for the

```
procedure unmark(u : node) :=
    avail(u) ← true
    foreach w ∈ B(u) do
        B(u) ← B(u) − {w}
        if ¬avail(w) then unmark(w)
    return
```

Algorithm 2: The procedure to make marked vertices available again as extensions to the path that is currently computed in procedure cycle

```
procedure find_all_cycles :=
  cycles_found ← ∅
  path ← the empty stack
  s ← 1
  while s < |N| do
    Let S = (N_S, E_S) be the nontrivial, strongly connected component containing
      the smallest numbered node in the subgraph of G induced by {s, ... , |N|}.
    if N_S ≠ ∅ then Adj ← adjacency structure for S
                    s ← smallest numbered node in N_S
                    foreach v ∈ N_S do avail(v) ← true
                                       B(v) ← ∅
                    cycle(s, f)
                    s ← s + 1
               else s ← |N|
  return
```

Algorithm 3: The procedure to generate all cycles for a digraph $\mathcal{G} = (N, E)$

edges. This is a worklist algorithm which keeps track of the nodes that are processed and recursively builds up the adjacency structure. The function operates on a four-tuple ($\text{SET[ID]} \times \text{SET[ID]} \times \text{SET[ID}\times\text{SET[ID]]} \times \text{CONEQS}$) denoting respectively the nodes to process, the nodes already processed, the graph already found (coded in an adjacency structure), and the source equations.

$$\text{[graph-recurse]} \quad \frac{\varepsilon^* = \varepsilon_1^* \ \nu \equiv\{\sigma\} \ \varepsilon_2^*, \ \ calls_{\nu \ \equiv\{\sigma\}} = \mathcal{S}_2^{\nu}}{\begin{array}{l} f_{graph}(\{\nu, \nu_1^*\}, \mathcal{S}_1^{\nu}, \{\theta^*\}, \varepsilon^*) \ = \\ f_{graph}(\{\nu_1^*\} \cup \mathcal{S}_2^{\nu} \ / \ (\mathcal{S}_1^{\nu} \cup \{\nu\}), \mathcal{S}_1^{\nu} \cup \{\nu\}, \{\theta^*, (\nu, \mathcal{S}_2^{\nu})\}, \varepsilon^*) \end{array}}$$

$$\text{[graph-found]} \quad f_{graph}(\{\}, \mathcal{S}^{\nu}, \mathcal{S}^{\theta}, \varepsilon^*) = \mathcal{S}^{\theta}$$

Find the extended topological order of nodes of a graph. This is a worklist algorithm which recursively builds up the order. The order is also used to keep track of the nodes that are already processed. Edges are considered back- or cross edges when they are directed towards nodes that are already in the order, consequently their end-nodes are not added to the worklist. The function operates on a three-tuple ($\text{LIST[ID]} \times \text{LIST[ID]} \times \text{SET[ID}\times\text{SET[ID]]}$) denoting respectively the nodes to process, the nodes already ordered, and the graph (coded in an adjacency structure).

$$\text{[eto-ready]} \quad \mathcal{O}_{xt}([], \mathcal{L}, \mathcal{S}^{\theta}) = \mathcal{L}$$

$$\text{[eto-recurse]} \quad \frac{\begin{array}{c} \mathcal{S}^{\theta} = \{\theta_1^*, (\nu_1, \mathcal{S}_1^{\nu}), \theta_2^*\}, \\ \mathcal{S}_1^{\nu} \ / \ \{\nu_1, \nu_1^*, \nu_2^*\} = \{\nu_3^*\} \end{array}}{\mathcal{O}_{xt}([\nu_1, \nu_1^*], [\nu_2^*], \mathcal{S}^{\theta}) \ = \ \mathcal{O}_{xt}([\nu_1^*, \nu_3^*], [\nu_2^*, \nu_1], \mathcal{S}^{\theta})}$$

For all loops in the graph, place the loop-headers after the last element of their bodies in a given ordered list. The function expects an ordered list (usually, the extended topological order of the graph) and a set of loops: tuples of the form $\text{ID} \times \text{IDLIST}$ where the first denotes the loop-header and the second an ordered list of the nodes of its body. When there are no loops, no changes have to be made and we are ready.

$$\text{[change-empty]} \quad \mathcal{O}_{ch}(\mathcal{L}, \{\}) = \mathcal{L}$$

When for a given loop, the loop-header occurs before the last element of its body in the order, put it after that element in the order and remove the loop from the set of loops.

[do-change]
$$\begin{aligned}&\mathcal{O}_{ch}([\nu_1^*, \nu_1, \nu_2^*, \nu_2, \nu_3^*], \{(\nu_1, [\nu_4^*, \nu_2]), \zeta_2^*\})\\ &\quad = \mathcal{O}_{ch}([\nu_1^*, \nu_2^*, \nu_2, \nu_1, \nu_3^*], \{\zeta_2^*\})\end{aligned}$$

When for a given loop, the loop-header already occurs after the last element of its body (eg. due to a previous change), leave it there and remove the loop from the set of loops.

[dont-change]
$$\begin{aligned}&\mathcal{O}_{ch}([\nu_1^*, \nu_2, \nu_2^*, \nu_1, \nu_3^*], \{(\nu_1, [\nu_4^*, \nu_2]), \zeta_2^*\})\\ &\quad = \mathcal{O}_{ch}([\nu_1^*, \nu_2, \nu_2^*, \nu_1, \nu_3^*], \{\zeta_2^*\})\end{aligned}$$

As described before, the algorithm to find the loops is not algebraically specified but given as pseudocode. We present a dummy function here to keep the specification complete.

[loops-dummy]
$$f_{loops}(\mathcal{S}^\theta) = \{\}$$

5.7.1 Remarks

The method presented here was based on a method to find an optimal ordering on the equations in order to control the code size method which was partially described in [Amm90, Amm92]. However, these articles do not give a rationale for the approach and leave out some significant steps.

5.8 Control flow normalization

This section describes the specification that interconnects all previous specifications and provides the complete control flow normalization function (N) on $\mathfrak{Dhal}$ programs.

module Normalize
imports Dhal-syntax[4.4.1] CE-preprocess[5.4.1] CE-find[5.4.2] Resolution-order[5.7] Resolution[5.6]
exports
  context-free syntax
    N DHALPROGRAM $\rightarrow$ DHALPROGRAM
hiddens
  context-free syntax
    N_{stat} "(" DHALSTAT ")" $\rightarrow$ DHALSTAT
    N_{decl} "(" DHALDECLS ")" $\rightarrow$ DHALDECLS
equations

The complete control flow normalization process consists of preprocessing the program and normalizing its declarations and statements.

[norm-prog]
$$\frac{\wp(\textbf{program}\ \nu\{\delta^*\ \sigma\}) = \textbf{program}\ \nu'\{\delta_1^*\ \sigma_1\},\quad N_{decl}(\delta_1^*) = \delta_2^*,\ \ N_{stat}(\sigma_1) = \sigma_2}{N\ \textbf{program}\ \nu\{\delta^*\ \sigma\}\ =\ \textbf{program}\ \nu'\{\delta_2^*\ \sigma_2\}}$$

Normalizing a (series of) statement(s) involves extraction of its continuation equations, followed by solving those equations using the resolution method.

[norm-stat]
$$\frac{f_\varepsilon\ \sigma_1 = \varepsilon^*,\ \ \Sigma(\mathcal{O}\ \varepsilon^*, \varepsilon^*) = \sigma_2}{N_{stat}(\sigma_1)\ =\ \sigma_2}$$

Normalizing an empty declaration is trivial:

[norm-decl-empty] $$N_{decl}() =$$

Normalizing a procedure consists of normalizing its statements.

[norm-decl-proc] $$\frac{N_{stat}(\sigma_1) = \sigma_2,\;\; N_{decl}(\delta_1^*) = \delta_2^*}{N_{decl}(\mathbf{proc}\ \nu\ \mathcal{L}\{\sigma_1\}\ \delta_1^*)\ =\ \mathbf{proc}\ \nu\ \mathcal{L}\{\sigma_2\}\ \delta_2^*}$$

All other declarations are ignored (this allows future extensions of the $\mathfrak{Dhal}$ language).

[norm-decl-other] $$\frac{N_{decl}(\delta_1^*) = \delta_2^*}{N_{decl}(\delta\ \delta_1^*)\ =\ \delta\ \delta_2^*} \quad \textbf{otherwise}$$

Chapter 6

Variable aliasing

Aliasing occurs when there are two or more identifiers that refer to the same memory location. Thus, 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 [ADS91, LR91, MLR$^+$93]. To solve these problems, more research on both detection of aliasing and the integration of alias information into program analysis should, and has been, done (see for example [CBC93, LR92, LRZ93, Wei80, WL95]).

6.1 Aliasing of access paths

As an extension of basic variable names, we will use the notion of *access paths* to denote identifiers that refer to storage locations [LH88].

Definition 6.1 (Access path) An access path is a left-value expression that refers to a storage location and is constructed from basic program variable names, pointer indirection operators and structure field operators. □

For example, in the C programming language, access paths would typically include the '`*`' indirection- and the '`[ ]`', '`->`' and '`.`' field selection operators.

Using access paths, we can give a more precise definition of aliasing. Therefore, it is important to consider that in the presence of variable subscripting (array indexing) or pointer indirection, aliases become dynamic entities i.e. they can be changed during execution of the program. Furthermore, in the presence of structured variables (arrays and records) aliases not only exist between access paths that refer to exactly the same storage locations but also between access paths that refer to storage locations that intersect with each other.

Definition 6.2 (Aliasing) Two access paths ν_i and ν_j are called aliases at a given point in a program if and only if they refer to an overlapping storage location at that point. □

6.2 Types of aliasing

In classic data flow literature (see for example [Hec77, ASU86]), the alias relation is considered an equivalence relation on access paths. We show that this assumption is incorrect and leads to inaccurate (but conservative) results. We propose a subdivision of the alias relations into four distinct types and prove that only one of these is an equivalence relation. Our distinction is 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.

Proposition 6.1 The alias relations described above have the following properties:

a) must alias relations are reflexive and transitive;

b) maybe alias relations are not reflexive and not transitive;

c) complete alias relations are symmetric;

d) partial alias relations are asymmetric.

□

Proof Let p, q and r be distinct access paths and e_1, e_2 be distinct execution paths:

a) *reflexive:* By definition holds at a given point for *all* execution paths that p refers to a storage location that overlaps with the location that itself refers to.

   transitive: Let (p, q) and (q, r) be must aliases at a given point. Then, for *all* execution paths, p refers to a storage locations that overlaps with the location that q refers to and q refers to a storage location that overlaps with the location that r refers to. Consequently, for *all* execution paths, p refers to a storage location that overlaps with the location that r refers to. So (p, r) is a must alias.

b) *not reflexive:* Suppose the relation is reflexive. Then (p, p) is a maybe alias, so at a given point it cannot be determined for all *all* execution paths that p refers to a storage locations that overlaps with the location that itself refers to (contradiction).

   not transitive: Let (p, q) and (q, r) be maybe aliases. Then e_1 might determine that p refers to a storage location that overlaps with the location that q refers to and e_2 might determine that q refers to a storage location that overlaps with the location that r refers to. Consequently, it cannot be determined that p refers to a storage location that overlaps with the location that r refers to.

c) Let (p, q) be complete aliases. Then p refers to exactly the same storage location as q. Consequently, q refers to exactly the same storage location as p. Therefore (q, p) are complete aliases.

d) Let (p, q) be partial aliases and suppose that the relation is symmetric. Then (q, p) is a partial alias, so p refers to a storage location which includes the location that q refers to and q refers to a storage location which includes the location that p refers to (contradiction). □

Corollary 6.2 The complete must alias relation is an equivalence relation on access paths. □

Proof Complete implies symmetric; must implies reflexive and transitive. □

Thus, the assumption that alias relations are equivalence relations (eg. [Hec77, ASU86]) is incorrect. The result of this assumption is that the transitive closure of the alias relations is used for too many cases, thus too many access paths are considered to be aliases. Consequently, the computed solution contains a *conservative* error with respect to the correct solution. Using our proposed distinction, these errors do not occur which is important for the step described in the next section: the precise integration of alias information in data flow analysis.

6.3 Integration of alias information

In order to solve a data flow problem for a program, we need to know the side effects of the program statements in terms of defined and used variables and we need to know which variables are potential aliases. Traditional data flow analysis approaches tried to solve all these at the same time. Banning was the first who suggested the approach of decomposing inter-procedural side effect analysis into two subproblems [Ban78, Ban79]:

1. the *detection* of potential aliases;

2. *alias-free* side effect analysis.

For many important data flow problems, these two components can be solved independently and their results can be combined later to produce an overall solution.

However, Bannings decomposition technique cannot be applied to all programs. Incorrect solutions occur when side effect analysis is performed separately from alias analysis in the presence of general pointers [CBC93]. These errors originate from the fact that, using pointers, the storage location that a given reference variable points to, can be changed in the body of a procedure. As a result, a statement which seems to be a definition of that variable, can in fact be the definition of another variable; similar for statements that seem to use such a variable (see example 6.3.1).

6.3.1 Representative parameters

Choi et all. [CBC93] present an elegant solution to adapt Bannings decomposition technique [Ban79] to the presence of general pointers. They adopt Bannings technique and extend it with the introduction of a so-called *representative parameter* r_i for each ith parameter of a procedure. This representative parameter is considered an alias of the actual parameter at call time (i.e. it refers to the same storage location as the actual parameter does when it enters the procedure). When the alias information is propagated through the procedure (and all references to variables local to the procedure are discarded) the procedure's side effects can be found in terms of its representative parameters and global variables.

Example 6.3.1 This example shows the error that is made in Bannings side-effect computation in the presence of general pointers and how it can be prevented using representative parameters. Consider the following piece of program code.

```
s1  void f(int *a, int *b) {
s2    int *tmp;
s3    tmp = a;
s4    a = b;
s5    b = tmp;
s6    *a = *b;
s7  }
s8  f(x, y)
```

With Bannings side effect analysis the *DEF* and *USE* sets of statement s_8 are as follows: $\mathsf{def}(s_8) = \{x\}$, $\mathsf{use}(s_8) = \{y\}$. However, if we follow the pointer dereferencing, we see that actually y is defined and x is used. When representative parameters r_1 and r_2 are introduced as aliases for respectively $*a$ and $*b$ during the alias detection, there are the following aliases at the exit of the procedure: $(*a, r_2), (*b, r_1)$ and $(*tmp, r_1)$. Consequently, we can conclude that $\mathsf{def}(s_8) = \{y\}$, $\mathsf{use}(s_8) = \{x\}$.

□

6.3.2 Alias Propagation

We propose to use a combination of the approaches of Bannings and Choi et all. and decompose the process of data flow analysis into three consecutive steps:

1. alias detection combined with the introduction of representative parameters;
2. *propagation* of alias information in $\mathfrak{Dhal}$ programs;
3. alias-free data flow analysis on $\mathfrak{Dhal}$ programs.

The remainder of this section concentrates on the second step: the propagation of alias information in $\mathfrak{Dhal}$ programs.

A straightforward alias propagation technique

There are several schemes possible for the propagation of aliases in $\mathfrak{Dhal}$ programs. One of the most straightforward schemes is propagating the aliases in both the *def* and the *use* sets of the side effect statements of $\mathfrak{Dhal}$. According to Burke and Choi this results in imprecise (but conservative) data flow information since *may aliases* will be considered in too many cases when constructing def-use or def-def chains [BC92]. A def-use chain of a access path v is constructed from statement s_i to s_j iff:

1. $v \in \mathsf{def}(s_i) \cap \mathsf{use}(s_j)$
2. there exists a path $P_{i,j} : s_i \twoheadrightarrow s_j$ in the control flow graph along which v is not killed.

Similarly, a def-def chain of a access path v is constructed from statement s_i to s_j iff:

1. $v \in \mathsf{def}(s_i) \cap \mathsf{def}(s_j)$
2. there exists a path $P_{i,j} : s_i \twoheadrightarrow s_j$ in the control flow graph along which v is not killed.

In the remainder of this chapter we will assume that the second condition of these definitions is fulfilled for the cases that we examine, so we will only consider the first one.

Example 6.3.2 This example shows the straightforward propagation of aliases in a $\mathfrak{Dhal}$ program and the error that is made using this approach. Consider the following input program and the propagation of its aliases into the sets of defined and used variables. Statement s_1 defines a complete must alias, and statement s_2 defines a complete may alias. Both these statements are replaced by skip statements during alias propagation.

	input program	aliases propagated in def & use sets
s_1	$\leftrightarrow \{(d,e)\};$	$\tau;$
s_2	$\leftrightsquigarrow \{(a,b),(a,c)\};$	$\tau;$
s_3	$\{a\} \ltimes \{\};$	$\{a,b,c\} \ltimes \{\};$
s_4	$\{b\} \ltimes \{\};$	$\{b,a\} \ltimes \{\};$
s_5	$\{\} \ltimes \{c\};$	$\{\} \ltimes \{c,a\};$
s_6	$\{c\} \ltimes \{\};$	$\{c,a\} \ltimes \{\};$
s_7	$\{d\} \ltimes \{\};$	$\{d,e\} \ltimes \{\};$
s_8	$\{e\} \ltimes \{\}$	$\{d,e\} \ltimes \{\}$

Analysis of the propagated program yields a def-use chain for a from s_4 to s_5 because it was defined at s_4 (as an aliased definition) and used at s_5 (as an aliased use). The presence of such a chain indicates that the value of a variable which is defined at one end will be used at the other. However, in the example program, this chain is spurious since we cannot determine whether the value of a that is used at s_5 is really defined at s_4 (note that maybe alias relations are not transitive and in the example $(b,c) \notin \leftrightsquigarrow$). Similarly, there is a spurious def-def chain for a from s_4 to s_6 which (incorrectly) indicates that the value of a is defined at s_4 and redefined at s_6. □

An alias propagation technique due to Banning

Bannings technique for comparing the side effects of two statements [Ban79] also presents an alias propagation technique. This technique does not suffer from the spurious def-use chains that occur with straightforward alias propagation. Banning distinguishes the following side effects of a statement s:

- $\mathsf{dmod}(s)$ is the set of access paths that are defined in s without aliases factored in
- $\mathsf{mod}(s)$ is the set of access paths that are defined in s with aliases factored in
- $\mathsf{dref}(s)$ is the set of access paths that are used in s without aliases factored in
- $\mathsf{ref}(s)$ is the set of access paths that are used in s with aliases factored in

(note that Banning uses the terms "modified" for "defined" and "referred" for "used"). Next, he defines a def-use chain for v between s_1 and s_2 if $v \in \mathsf{dmod}(s_1) \cap \mathsf{ref}(s_2)$ (or similar $v \in \mathsf{mod}(s_1) \cap \mathsf{dref}(s_2)$). This eliminates the spurious chain from s_4 to s_5 in the previous example. Similarly, he defines a def-def chain for v between s_1 and s_2 if $v \in \mathsf{dmod}(s_1) \cap \mathsf{mod}(s_2)$ (or similar $v \in \mathsf{mod}(s_1) \cap \mathsf{dmod}(s_2)$) which eliminates the spurious chain from s_4 to s_6 in that example.

Bannings approach seems to be overlooked in most of the subsequent literature since the majority used the straightforward scheme described earlier. Finally, Burke and Choi described the occurrence of spurious def-use and def-def chains and proposed a solution [BC92]. Their technique consists of two mechanisms: first, they propagate alias information only into the *def* sets and do not propagate aliasing in the *use* sets at all. This step in effect defines $\mathsf{use}(s)$ to be $\mathsf{dref}(s)$ from Bannings approach. Second, they tag the elements of the *def* sets with boolean values, thereby dividing them into two subsets: (1) the original (alias-free) definitions and those definitions that result from *must aliases*, and (2) the definitions that result from *may aliases*. This reduces Bannings intersection of $\mathsf{dmod}(s_1)$ and $\mathsf{mod}(s_2)$ to a simple bit test.

Our alias propagation technique

We present an alias propagation technique which combines the techniques of Banning and Burke and Choi. Our alias propagation function p does not propagate the aliases in the *use* sets but we will actually separate the two subsets of defined access paths: we will refer to the first as the set of *direct* definitions ($\mathsf{ddef}(s)$) and to the second as the set of *induced* definitions ($\mathsf{idef}(s)$). As a result of this distinction, the language p$\mathfrak{Dhal}$ (*propagated* $\mathfrak{Dhal}$) uses slightly revised statements to describe side effects since we need to relate two sets of defined variables to a set of used variables. We recall that the syntax for p$\mathfrak{Dhal}$ was described in Section 4.4.3.

We define a def-use chain for v between s_1 and s_2 if $v \in (\mathsf{ddef}(s_1) \cup \mathsf{idef}(s_1)) \cap \mathsf{use}(s_2)$. Furthermore, we define a def-def chain for v between s_1 and s_2 if $v \in (\mathsf{ddef}(s_1) \cap \mathsf{ddef}(s_2)) \cup (\mathsf{ddef}(s_1) \cap \mathsf{idef}(s_2)) \cup (\mathsf{idef}(s_1) \cap \mathsf{ddef}(s_2))$ (i.e. at least one of the definitions of v is a direct definition).

module Alias-propagation
imports Structured-dhal-syntax$^{4.4.2}$ Propagated-dhal-syntax$^{4.4.3}$
exports
  context-free syntax
    p "(" DHALPROGRAM ")" → DHALPROGRAM
hiddens
  sorts AENV DAENV SAENV
  context-free syntax
    SET[ID×ID] "," SET[ID×ID] "," SET[ID×ID] "," SET[ID×ID] → AENV
    "(" DHALDECLS "," AENV ")" → DAENV
    p_{decl} DAENV → DAENV
    "(" DHALSTAT "," AENV ")" → SAENV
    p_{stat} SAENV → SAENV
    ali "(" SET[ID] "," SET[ID×ID] "," SET[ID] ")" → SET[ID]
  variables
    $\mathcal{E}^{\mathcal{A}}$ $[0-9\,']*$ → AENV
equations

The p_{decl} and p_{stat} functions propagate the aliases in respectively the declarations and statements in a $\mathfrak{Dhal}$ program. The first argument of these functions is respectively a declaration or a statement and the second argument is a four-tuple (SET[ID×ID] × SET[ID×ID] × SET[ID×ID] × SET[ID×ID]) denoting respectively the current must complete aliases, may complete aliases, must partial aliases, and may partial aliases. Propagation of aliases in a program distributes over the declarations and the statements in the body:

[pa-prog] $$\frac{p_{decl}\ (\delta_1^*, \{\}, \{\}, \{\}, \{\}) = (\delta_2^*, \mathcal{E}_1^{\mathcal{A}}),\ \ p_{stat}\ (\sigma_1, \mathcal{E}_1^{\mathcal{A}}) = (\sigma_2, \mathcal{E}_2^{\mathcal{A}})}{p(\textbf{program}\ v\{\delta_1^*\ \sigma_1\})\ =\ \textbf{program}\ v\{\delta_2^*\ \sigma_2\}}$$

The empty declaration needs no propagation:

[pa-decl-empty] $$p_{decl}\ (\,,\ \mathcal{E}^{\mathcal{A}}) = (\,,\ \mathcal{E}^{\mathcal{A}})$$

Alias propagation in procedure declarations distributes over the statements in the body. Note that procedure declarations cannot change the alias definitions for their successors

[pa-decl-proc] $$\frac{p_{stat}\ (\sigma_1,\ \mathcal{E}_1^{\mathcal{A}}) = (\sigma_2,\ \mathcal{E}_2^{\mathcal{A}})}{p_{decl}\ (\textbf{proc}\ \nu\ \mathcal{L}\{\sigma_1\},\ \mathcal{E}_1^{\mathcal{A}})\ =\ (\textbf{proc}\ \nu\ \mathcal{L}\{\sigma_2\},\ \mathcal{E}_1^{\mathcal{A}})}$$

All other declarations are ignored. This allows future extensions of the $\mathfrak{Dhal}$ language such as alias declarations etcetera.

[pa-decl-other] $$p_{decl}\ (\delta_1,\ \mathcal{E}^{\mathcal{A}}) = (\delta_1,\ \mathcal{E}^{\mathcal{A}})\quad \textbf{otherwise}$$

Propagation of aliases in the composition of two or more declarations involves propagation of the aliases in the first (eventually changing existing alias definitions) followed by propagation of the (new) aliases in the rest.

[pa-decl-comp] $$\frac{p_{decl}\ (\delta_1,\ \mathcal{E}_1^{\mathcal{A}}) = (\delta_3,\ \mathcal{E}_2^{\mathcal{A}}),\ \ p_{decl}\ (\delta_2^+,\ \mathcal{E}_2^{\mathcal{A}}) = (\delta_4^*,\ \mathcal{E}_3^{\mathcal{A}})}{p_{decl}\ (\delta_1\ \delta_2^+,\ \mathcal{E}_1^{\mathcal{A}})\ =\ (\delta_3\ \delta_4^*,\ \mathcal{E}_3^{\mathcal{A}})}$$

Alias propagation in alias defining statements consists of the redefinition of the corresponding relations in the alias environment. The next four equations do this for respectively must complete, may complete, must partial, and may partial aliases.

[pa-stat-muscom] $p_{stat}\ (\leftrightarrow \mathcal{A},\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A}_4) = (\tau,\ \mathrm{sym}(\mathcal{A}),\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A}_4)$
[pa-stat-maycom] $p_{stat}\ (\leftrightsquigarrow \mathcal{A},\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A}_4) = (\tau,\ \mathcal{A}_1,\ \mathrm{sym}(\mathcal{A}),\ \mathcal{A}_3,\ \mathcal{A}_4)$
[pa-stat-muspar] $p_{stat}\ (\rightarrow \mathcal{A},\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A}_4) = (\tau,\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A},\ \mathcal{A}_4)$
[pa-stat-maypar] $p_{stat}\ (\rightsquigarrow \mathcal{A},\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A}_4) = (\tau,\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A})$

In order to propagate aliases in statements that describe side effects, we compute

- the set of direct defined access paths ($\mathcal{S}_3^{\nu}$) which is the union of the original definitions ($\mathcal{S}_1^{\nu}$) and the complete- and partial must aliases of those definitions, and
- the set of inductively defined access paths ($\mathcal{S}_4^{\nu}$) which is the set of complete- and partial maybe aliases of these definitions.

A statement sometimes results in the direct and the induced definition of the same access path. These induced definitions which duplicate direct definitions are discarded since direct definitions contains more information because they stem from original definitions or must aliases. Finally, the side effect statement is replaced by one that relates these two sets to the original set of used access paths.

[pa-stat-se] $$\frac{\begin{array}{c} ali(\mathcal{S}_1^{\nu},\ \mathcal{A}_1,\ \mathcal{S}_1^{\nu}) \cup ali(\mathcal{S}_1^{\nu},\ \mathcal{A}_3,\ \mathcal{S}_1^{\nu}) = \mathcal{S}_3^{\nu}, \\ (ali(\mathcal{S}_3^{\nu},\ \mathcal{A}_2,\ \{\}) \cup ali(\mathcal{S}_3^{\nu},\ \mathcal{A}_4,\ \{\}))\ /\ \mathcal{S}_3^{\nu} = \mathcal{S}_4^{\nu} \end{array}}{\begin{array}{l} p_{stat}\ (\mathcal{S}_1^{\nu} \ltimes \mathcal{S}_2^{\nu},\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A}_4)\ = \\ ((\mathcal{S}_3^{\nu},\ \mathcal{S}_4^{\nu}) \ltimes \mathcal{S}_2^{\nu},\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A}_4) \end{array}}$$

Alias propagation of conditional and iteration statements recurses into their body statements. The alias definitions found in either the *then* or *else* parts of an conditional statement are united and used to analyze the rest of the program.

[pa-stat-if] $$\frac{\begin{array}{c} p_{stat}\ (\sigma_1,\ \mathcal{E}_1^{\mathcal{A}}) = (\sigma_1',\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \mathcal{A}_4), \\ p_{stat}\ (\sigma_2,\ \mathcal{E}_1^{\mathcal{A}}) = (\sigma_2',\ \mathcal{A}_5,\ \mathcal{A}_6,\ \mathcal{A}_7,\ \mathcal{A}_8), \\ \mathcal{E}_2^{\mathcal{A}} = \mathcal{A}_1 \cup \mathcal{A}_5,\ \mathcal{A}_2 \cup \mathcal{A}_6,\ \mathcal{A}_3 \cup \mathcal{A}_7,\ \mathcal{A}_4 \cup \mathcal{A}_8 \end{array}}{\begin{array}{l} p_{stat}\ (\textbf{if}\ \mathcal{S}^{\nu}\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\},\ \mathcal{E}_1^{\mathcal{A}})\ = \\ (\textbf{if}\ \mathcal{S}^{\nu}\ \textbf{then}\{\sigma_1'\}\ \textbf{else}\{\sigma_2'\},\ \mathcal{E}_2^{\mathcal{A}}) \end{array}}$$

[pa-stat-while] $$\frac{p_{stat}\ (\sigma_1,\ \mathcal{E}_1^{\mathcal{A}}) = (\sigma_1',\ \mathcal{E}_2^{\mathcal{A}})}{p_{stat}\ (\mathbf{do}\{\sigma_1\}\ \mathbf{while}\ \mathcal{S}^{\nu},\ \mathcal{E}_1^{\mathcal{A}})\ =\ (\mathbf{do}\{\sigma_1'\}\ \mathbf{while}\ \mathcal{S}^{\nu},\ \mathcal{E}_2^{\mathcal{A}})}$$

Propagation of aliases in the composition of two statements involves propagation of the aliases in the first (eventually changing existing alias definitions) followed by propagation of the (new) aliases in the second.

[pa-stat-comp] $$\frac{p_{stat}\ (\sigma_1,\ \mathcal{E}_1^{\mathcal{A}}) = (\sigma_1',\ \mathcal{E}_2^{\mathcal{A}}),\ \ p_{stat}\ (\sigma_2,\ \mathcal{E}_2^{\mathcal{A}}) = (\sigma_2',\ \mathcal{E}_3^{\mathcal{A}})}{p_{stat}\ (\sigma_1;\ \sigma_2,\ \mathcal{E}_1^{\mathcal{A}})\ =\ (\sigma_1';\ \sigma_2',\ \mathcal{E}_3^{\mathcal{A}})}$$

All other statements have no influence and are literally copied.

[pa-stat] $$p_{stat}\ (\sigma,\ \mathcal{E}^{\mathcal{A}}) = (\sigma,\ \mathcal{E}^{\mathcal{A}})\quad \mathbf{otherwise}$$

The $ali(\mathcal{S}^{\nu}_{in}, \mathcal{A}_{in}, \mathcal{S}^{\nu}_{st})$ function is used to find the set of aliases that are induced by a given alias relation ($\mathcal{A}_{in}$) on a given input set ($\mathcal{S}^{\nu}_{in}$). The third parameter is used to store intermediate results (and to define the set of aliases to start with) When the alias relation is empty, the *ali* function delivers the stored set of aliases.

[ali-empty] $$ali(\mathcal{S}^{\nu}_1,\ \{\},\ \mathcal{S}^{\nu}_2) = \mathcal{S}^{\nu}_2$$

If there is an element in the input set which has an alias according to the relation, that alias is added to the stored set:

[ali-fnd] $$ali(\{\nu_1^*,\ \nu_2,\ \nu_3^*\},\ \{(\nu_2,\ \nu_4),\ \alpha^*\},\ \mathcal{S}^{\nu}) = ali(\{\nu_1^*,\ \nu_2,\ \nu_3^*\},\ \{\alpha^*\},\ \mathcal{S}^{\nu} \cup \{\nu_4\})$$

Otherwise, the first element of the alias relation is deleted and the search continues until the relation is empty:

[ali-other] $$ali(\mathcal{S}^{\nu}_1,\ \{\alpha,\ \alpha^*\},\ \mathcal{S}^{\nu}_2) = ali(\mathcal{S}^{\nu}_1,\ \{\alpha^*\},\ \mathcal{S}^{\nu}_2)\quad \mathbf{otherwise}$$

Example 6.3.3 This example shows the propagation of aliases in a $\mathfrak{Dhal}$ program using our alias propagation technique. Furthermore, it shows that the error which occurred using the straightforward propagation does not occur using this approach. Consider the same input program as example 6.3.2 and the result of the alias propagation function *p*:

	input program	result of *p*
s_1	$\leftrightarrow \{(d, e)\};$	$\tau;$
s_2	$\leadsto \{(a, b), (a, c)\};$	$\tau;$
s_3	$\{a\} \ltimes \{\};$	$(\{a\}, \{b, c\}) \ltimes \{\};$
s_4	$\{b\} \ltimes \{\};$	$(\{b\}, \{a\}) \ltimes \{\};$
s_5	$\{\} \ltimes \{c\};$	$(\{\}, \{\}) \ltimes \{c\};$
s_6	$\{c\} \ltimes \{\};$	$(\{c\}, \{a\}) \ltimes \{\};$
s_7	$\{d\} \ltimes \{\};$	$(\{d, e\}, \{\}) \ltimes \{\};$
s_8	$\{e\} \ltimes \{\}$	$(\{e, d\}, \{\}) \ltimes \{\}$

Analysis of the propagated program yields a def-use chain for c from s_3 to s_5 because c is defined at s_3 (as an aliased definition) and used at s_5 (as an original use). Furthermore, there is a def-def chain for c from s_3 to s_6 because c is defined at s_3 (as an aliased definition) and redefined at s_6 (as an original definition). Similarly, there are def-def chains for d and e from s_7 to s_8 because their values are defined at s_7 and redefined at s_8. □

6.4 Detection of aliases

In depth treatment of the detection (or calculation) of aliases is beyond the scope of this thesis. Such detection should be performed during the translation from the source language to $\mathfrak{Dhal}$ since the source language contains the language constructs that introduce aliases.

Landi and Ryder proved that the precise calculation of aliases in the presence of either multi level pointers, single level pointers and structures or single level pointers and pointer reference parameters is $\mathcal{NP}$-hard or co-$\mathcal{NP}$-hard [LR91]. Consequently, a lot of work has been done on the safe approximation of alias relations in the presence of multiple levels of indirection (eg. [HPR89, LR92, CBC93, LRZ93]) and future work should definitely include investigation of these techniques and of ways to include them in our architecture.

Furthermore, there is a relation between detection of aliases and a relatively new technique called *points-to analysis* [EGH94] which should be investigated. Points-to analysis is a technique which computes a storage model for a computer program using abstract locations. From this storage model all kinds of information can be derived, possibly including the alias relations between access paths (see for example [EGH94, Ruf95, Ste96]).

Chapter 7

Variable dependence tracking

We will now present a formal specification of a data flow analysis algorithm solving the variable dependence tracking problem described earlier. The specification serves as an example of the specification of data flow analysis algorithms using the proposed data flow framework, the data flow representation language $\mathfrak{Dhal}$ and the algebraic specification formalism ASF+SDF.

7.1 Introduction

Variable dependence tracking is a data flow problem that can be used to support program understanding and reverse engineering. It gives answers to questions like: "Given a variable ν, what other variables are either used to define the value of ν or have a value which depends in some way on ν?".

Variable dependence tracking is a so-called *closure problem*. Therefore, the data flow analysis algorithm to compute these dependencies cannot find the solution in a single pass over the source code but it needs to perform a *fixed point* computation. This means that the algorithm iteratively propagates the dependence information over the code until no changes take place any more.

7.2 Formal Specification

module Var-Dependence
imports Propagated-dhal-syntax$^{4.4.3}$
exports
  context-free syntax
    vdep "(" DHALPROGRAM "," SET[ID] ")" $\rightarrow$ SET[ID]
hiddens
  context-free syntax
    vdep_{decl} "(" DHALDECLS "," SET[ID] ")" $\rightarrow$ SET[ID]
    vdep_{stat} "(" DHALSTAT "," SET[ID] ")" $\rightarrow$ SET[ID]
    $\mathit{fix}_{\mathsf{vdep}_{stat}}$ "(" DHALSTAT "," SET[ID] ")" $\rightarrow$ SET[ID]
equations

The variable dependencies of a program are the variable dependencies of the declarations and (the fixed point of) the variable dependencies of the program body.

[vdep-prog] $\mathsf{vdep}(\textbf{program}\ \nu\{\delta_1^*\ \sigma_1\}, \mathcal{S}^\nu) = \mathsf{vdep}_{decl}(\delta_1^*, \mathcal{S}^\nu) \cup \mathit{fix}_{\mathsf{vdep}_{stat}}(\sigma_1, \mathcal{S}^\nu)$

An empty declaration does not add variable dependencies.

[vdd-empty] $$\mathsf{vdep}_{decl}(, \mathcal{S}^\nu) = \mathcal{S}^\nu$$

The variable dependencies of a procedure declaration are (the fixed point of) the variable dependencies of the procedure body.

[vdd-proc] $$\mathsf{vdep}_{decl}(\textbf{proc}\ \nu\ \mathcal{L}\{\sigma\}, \mathcal{S}^\nu) = \mathit{fix}_{\mathsf{vdep}_{stat}}(\sigma, \mathcal{S}^\nu)$$

All other declarations do not add variable dependencies.

[vdd-other] $$\mathsf{vdep}_{decl}(\delta, \mathcal{S}^\nu) = \mathcal{S}^\nu \quad \textbf{otherwise}$$

The variable dependencies of a composition of declarations are the variable dependencies of the first declaration and the variable dependencies of the rest of the declarations.

[vdd-comp] $$\mathsf{vdep}_{decl}(\delta_1\ \delta_2^+, \mathcal{S}^\nu) = \mathsf{vdep}_{decl}(\delta_1, \mathcal{S}^\nu) \cup \mathsf{vdep}_{decl}(\delta_2^+, \mathcal{S}^\nu)$$

The fixed point computation of variable dependencies continues when the dependencies did changed during the last pass.

[fp-vds-loop] $$\frac{\mathsf{vdep}_{stat}(\sigma, \mathcal{S}_1^\nu) = \mathcal{S}_2^\nu,\ \ \mathcal{S}_1^\nu \neq \mathcal{S}_2^\nu}{\mathit{fix}_{\mathsf{vdep}_{stat}}(\sigma, \mathcal{S}_1^\nu)\ =\ \mathit{fix}_{\mathsf{vdep}_{stat}}(\sigma, \mathcal{S}_2^\nu)}$$

The fixed point computation of variable dependencies ends when the dependencies did not changed during the last pass.

[fp-vds-end] $$\frac{\mathsf{vdep}_{stat}(\sigma, \mathcal{S}_1^\nu) = \mathcal{S}_2^\nu,\ \ \mathcal{S}_1^\nu = \mathcal{S}_2^\nu}{\mathit{fix}_{\mathsf{vdep}_{stat}}(\sigma, \mathcal{S}_1^\nu)\ =\ \mathcal{S}_1^\nu}$$

If any of the variables in the variable dependency set are used or defined in a side-effect statement, then all variables that are used or defined in that statement are added to the dependency set.

[vds-se] $$\frac{\mathcal{S}^\nu \cap (\mathcal{S}_1^\nu \cup \mathcal{S}_2^\nu \cup \mathcal{S}_3^\nu)\ \neq\ \{\}}{\mathsf{vdep}_{stat}((\mathcal{S}_1^\nu, \mathcal{S}_2^\nu) \ltimes \mathcal{S}_3^\nu, \mathcal{S}^\nu)\ =\ \mathcal{S}^\nu \cup \mathcal{S}_1^\nu \cup \mathcal{S}_2^\nu \cup \mathcal{S}_3^\nu}$$

If any of the variables in the variable dependency set are used or defined in a procedure side-effect statement, then all variables that are used or defined in that statement are added to the dependency set.

[vds-proc] $$\frac{\mathcal{S}^\nu \cap (\{\nu^*\} \cup \mathcal{S}_1^\nu \cup \mathcal{S}_2^\nu)\ \neq\ \{\}}{\mathsf{vdep}_{stat}((\mathcal{S}_1^\nu, \mathcal{S}_2^\nu) \ltimes \nu\ [\nu^*], \mathcal{S}^\nu)\ =\ \mathcal{S}^\nu \cup \mathcal{S}_1^\nu \cup \mathcal{S}_2^\nu}$$

The variable dependencies of an *if-then-else* statement are the variable dependencies of the *then* branch and the variable dependencies of the *else* branch.

[vds-if] $\mathsf{vdep}_{stat}(\textbf{if}\ \mathcal{S}^\nu\ \textbf{then}\{\sigma_1\}\ \textbf{else}\{\sigma_2\}, \mathcal{S}^\nu) = \mathsf{vdep}_{stat}(\sigma_1, \mathcal{S}^\nu) \cup \mathsf{vdep}_{stat}(\sigma_2, \mathcal{S}^\nu)$

The variable dependencies of a *while* statement are the fixed point of the variable dependencies of the body.

[vds-while] $$\mathsf{vdep}_{stat}(\mathbf{do}\{\sigma\}\ \mathbf{while}\ \mathcal{S}_1^{\gamma}, \mathcal{S}^{\nu}) = \mathit{fix}_{\mathsf{vdep}_{stat}}(\sigma, \mathcal{S}^{\nu})$$

All other statements do not add variable dependencies.

[vds] $$\mathsf{vdep}_{stat}(\sigma, \mathcal{S}^{\nu}) = \mathcal{S}^{\nu} \quad \textbf{otherwise}$$

The variable dependencies of a composition of statements are the variable dependencies of the second statement based on the variable dependencies of the first statement.

[vds-comp] $$\mathsf{vdep}_{stat}(\sigma_1; \sigma_2, \mathcal{S}^{\nu}) = \mathsf{vdep}_{stat}(\sigma_2, \mathsf{vdep}_{stat}(\sigma_1, \mathcal{S}^{\nu}))$$

Chapter 8

Concluding remarks

We have developed, and algebraically specified, a software architecture for the specification and execution of data flow analysis. This framework is language independent since it separates the source language from the *data flow representation language*: the language on which the data flow analysis is performed. This separation has three 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, developing data flow analysis algorithms becomes so straightforward that two (or more) algorithms can easily be combined into a single algorithm. This improves the global efficiency of the program analysis since only a single run is needed instead of separate runs for each (sub-)algorithm followed by a run to combine the results. Furthermore, for some combinations of algorithms, the precision of the overall solution may be improved since they can now make use each others intermediate results. Third, a data flow representation language like ours might form a suitable communication platform for information exchange on data flow analysis algorithms and their merits.

The proposed framework contains two generic components that facilitate transformations commonly found in data flow analysis algorithms at the level of the data flow representation language. We will summarize the advantages of these components below.

Control flow normalization This is a transformation on programs where unstructured cycles in the flow of control are converted (or *normalized*) into structured cycles. Control flow normalization in general has several advantages for data flow analysis. First, it reduces the number of syntactical constructs which must be treated during analysis. Second, the normalized program has a much simpler semantics, i.e., direct semantics instead of continuation semantics, which reduces the complexity of any program analysis that models this semantics (such as data flow analysis). Third, the specific method of control flow normalization used in this thesis makes it unnecessary to perform interval analysis since the structure of the program becomes so manifest in the syntax tree of the program that all interval information can be extracted directly.

Furthermore, there are also considerable benefits from control flow normalization on the level of the data flow representation language. First, such a generic solution enables us to define a more expressive input language which makes it an easier abstraction target for the users (i.e. they can now use $\mathfrak{Dhal}$, a language with both structured and unstructured flow of control, instead of s$\mathfrak{Dhal}$, a language with only structured control flow). Second, generic control flow normalization reliefs the users from implementing and performing this process themselves before or during the abstraction from source language to s$\mathfrak{Dhal}$.

Alias handling Aliasing occurs when there are two or more identifiers that refer to the same memory location. Thus, 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. In traditional data flow analysis literature, the alias relation is considered an equivalence relation on access paths. We have shown that this assumption is incorrect and leads to inaccurate (but conservative) results. We propose a subdivision of the alias relations into four distinct types and prove that only one of these is an equivalence relation. Furthermore, we describe an alias propagation technique based on this distinction which results in a more precise side effect analysis.

Future work

Application of the framework requires the definition of an abstraction component for the language to be analyzed. Future work includes the definition of these abstractions for a number of programming languages that are currently being used and reengineered (for example, Cobol, Fortran, C).

Although the data flow representation language supports procedures, no in-depth investigation on *interprocedural data flow analysis* (data flow analysis in the presence of procedures) was done. Algorithms for interprocedural analysis and techniques that are used for this kind of analysis (eg. call graph analysis) could be added to the framework as common components [Hec77, ASU86, CK89].

More work should be done on the incorporation of alias detection (or calculation) in the framework. This detection is proven to be $\mathcal{NP}$-hard or co-$\mathcal{NP}$-hard in the presence of multiple levels of indirection [LR91], so techniques for the safe approximation of these alias relations should be investigated (eg. [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 (see for example [EGH94, Ruf95, Ste96]).

Appendix A

The ASF+SDF formalism

ASF+SDF [BHK89a] is an algebraic specification formalism for the specification of the syntax and semantics of (programming) languages. It originates from the combination of the two formalisms ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism). The ASF formalism [BHK89b] is an algebraic specification formalism where *signatures* define the abstract syntax of functions and a set of *conditional equations* define the semantics of these functions. The SDF formalism [HHKR89] allows the simultaneous definition of lexical, concrete, and abstract syntax and implicitly defines a translation from text strings to abstract syntax trees. In ASF+SDF the ASF signature definitions have been replaced by SDF syntax definitions. The result is a standard mapping from text to abstract syntax trees which allows the association of semantics with the tree representation of text and introduces user-defined notation in specifications.

A.1 Modules

An ASF+SDF specification consists of a number of modules. Each module may define syntax rules and/or semantic rules. The notation used in the semantic rules depends on the definition of syntax rules. A module can use other modules of the specification by importing them. The result of importing a module is that all visible names of the imported module become available in the importing module. A module can make its declarations invisible for other modules by hiding them. The overall structure of a module is:

> `imports` *Names-of-imported-modules*
> *Section**
> `priorities`
>   *Priority-relations-between-context-free-rules*
> `equations`
>   *Conditional-equations*

Each *Section* starts with one of the keywords `exports` or `hiddens`; `exports` makes all entities in the section visible to other modules, `hiddens` makes all entities in the section local to the module. A *Section* has the form:

> *exports-or-hiddens*
>   `sorts` *Names-of-sorts*
>   `lexical syntax`
>   *Rules-of-the-lexical-syntax*

```
context-free-syntax
    Rules-of-the-context-free-syntax
variables
    Names-of-variables
```

Each of the subsections sorts, lexical syntax, context-free syntax, and variables are optional.

A.2 Example specification: Booleans

The Booleans specification described in this section, serves as a simple example of an ASF+SDF specification. It defines the constants `true` and `false` and the functions *and* ('$\wedge$'), *or* ('$\vee$'), and *not* ('$\neg$'). The binary functions *and* and *or* are left-associative infix operators; the unary function *not* is a prefix operator. The brackets can be used to override the standard priorities.

module Booleans
imports Layout[C.1]
exports
  sorts BOOL
  context-free syntax
    true → BOOL
    false → BOOL
    BOOL "∨" BOOL → BOOL {**left**}
    BOOL "∧" BOOL → BOOL {**left**}
    ¬ BOOL → BOOL
    "(" BOOL ")" → BOOL {**bracket**}
  priorities
    ¬ BOOL → BOOL > BOOL "∧"BOOL → BOOL > BOOL "∨"BOOL → BOOL
  variables
    b [0 − 9 ′]∗ → BOOL
equations
  The usual boolean laws apply:

[disj1]	true ∨ b = true	[conj1]	true ∧ b = b	[neg1]	¬ false = true
[disj2]	false ∨ b = b	[conj2]	false ∧ b = false	[neg2]	¬ true = false

All specifications in this thesis are presented in a literate programming style (i.e. they contain their own documentation). The specifications are typeset using ToLaTeX [KV95, Vis95], a system which translates ASF+SDF specifications to LaTeX code. This system uses superscripts at module names to refer to the locations where these modules are defined. For example, the module Layout which is imported in the example above is defined in Section C.1.

A.3 Sorts

A sort is declared by listing its name in a `sorts` section of a module. A sort name should start with a capital letter and may be followed by letters, digits, hyphens ('-'), and underscores ('_'). There are four predefined sorts (`LAYOUT, IGNORE, REJECT, CHAR`) which may not be redeclared.

A.4 Lexical Syntax

The lexical syntax describes the low level structure of text using *lexical tokens*. A lexical token consists of a sort name (used to distinguish classes of tokens) and the actual text of the token. Lexical syntax definitions consist of a set of declarations for *lexical functions*. Each lexical function definition consists of a regular expression followed by '`->`' and a result sort. All functions with the same result sort together define the lexical syntax of tokens of that sort. Regular expressions may contain strings, sort names, character classes, and repetition operators ('`*`' and '`+`').

A.5 Context-free Syntax

The context-free syntax describes the concrete and abstract syntactic structure of sentences in a language. A context-free syntax definition consists of a set of declarations for *context-free functions*. Each context-free functions consists of zero or more literals, sort names, or lists followed by '`->`' and a result sort. All functions with the same result sort together define the alternatives for that sort.

A.5.1 Lists

Regularly, context-free syntax definitions require the description of the repetition of a syntactic notion or of list structures (with or without separators) containing a syntactic notion. ASF+SDF provides the following primitives for describing lists:

- $S*$ defines zero or more repetitions of sort S
- $S+$ defines one or more repetitions of sort S
- $\{S\ sep\}*$ defines zero or more repetitions of sort S separated by the literal *sep*
- $\{S\ sep\}+$ defines one or more repetitions of sort S separated by the literal *sep*.

Lists may be used in the left-hand side of a context-free function and in the right-hand side of a variable declaration. They may not be used as result sort in the right-hand side of a context-free function. This causes no restrictions since a new sort can be introduced representing a list. This new sort may then be used in the left-hand side of other context-free rules.

Example A.5.1 Using lists, the syntax of the declaration of identifiers in a Pascal-like language can be defined as follows:

sorts ID DECL TYPE
lexical syntax
  [a-z]+ → ID
context-free syntax
  var {ID ","}+ ":" TYPE → DECL
  integer → TYPE
  real → TYPE

□

A.6 Priorities

The context-free syntax defined in an ASF+SDF specification may be ambiguous: there are sentences which have more than one associated abstract syntax tree. This is solved by defining the priority or associativity of functions. ASF+SDF has three mechanisms for doing so:

1. Relative priorities of functions (defined in the '`priorities`' section). This is used to define which functions bind stronger than other functions.
2. Associativity of functions (defined with one of the attributes `left`, `right`, `assoc` or `non-assoc` following the function declaration). This is used to solve ambiguities introduced by nested occurrences of binary operators.
3. Associativity of groups of functions (defined in the '`priorities`' section). This is used to solve ambiguities introduced by functions with the same priority.

A.7 Variables

Variables are declared in the '`variables`' section of a module. This section consists of a list of *naming schemes* for variables together with their sort. A naming scheme is a regular expression like the ones used in the lexical syntax except that sorts are not allowed. A variable declaration may thus define an unlimited number of variables. The sort of a variable may be a simple sort, or a list.

A.8 Equations

Equations are used to add *semantics* to functions declared in the context-free syntax. Equations establish equalities between terms. The equality of two terms *S* and *T* is defined by an *unconditional equation* of the form:

[*TagId*] *S* = *T*

where *TagId* is a sequence of letters and/or digits starting with an uppercase letter or a digit, and *S* and *T* must be terms of the same sort. A *conditional equation* consists of a number of *conditions* (*premises*) and a conclusion. It can be written in three (syntactically different) ways:

```
1.    [TagId]   S = T  when C1, C2, ... , Cn

2.    [TagId]   C1, C2, ... , Cn ===> S = T

3.    [TagId]   C1, C2, ... , Cn
                ================
                     S = T
```

where *C1*, *C2*, ... , *C2* are conditions which may be either *positive* ('*S* = *T*'), or *negative* ('*S* != *T*'). Note that list sorts are not allowed as the sort of the term at the left-hand side or right-hand side of a condition or conclusion. Recall that this causes no restrictions since new sorts can be introduced to represents the lists.

A.9 ASF+SDF Meta–environment

The ASF+SDF Meta–environment [Kli93] is an interactive development environment for the algebraic specification of formal (programming) languages. It automatically generates programming environment components from formal language specifications. The SDF part of a specification is used to generate *lazy, incremental parsers* [Rek92] and *syntax-directed editors* [Koo92]. The ASF part of a specification is made *executable* by orienting the equations from left to right and interpreting them as *conditional rewrite rules* in a *term rewriting system* [Hen91, Wal91].

Appendix B

Import graph of the specification

Appendix C

Remainder of the specification

C.1 Layout

module Layout
exports
  lexical syntax
    "%%"~[\n]*[\n] → LAYOUT
    [␣\t\n] → LAYOUT

C.2 Naturals

module Naturals
imports Layout$^{C.1}$
exports
  sorts NAT-CON NAT
  lexical syntax
    [0-9]+ → NAT-CON
  context-free syntax
    NAT-CON → NAT
    NAT "+" NAT → NAT {**left**}
    "(" NAT ")" → NAT {**bracket**}
  variables
    n $[0-9']*$ → NAT
hiddens
  variables
    nc $[0-9']*$ → NAT-CON
    c $[0-9]*$ → CHAR
    x $[0-9]*$ → CHAR*
    y $[0-9]*$ → CHAR+
equations
  The usual equations for naturals apply. They are omitted to save space.

C.3 Identifiers

module Identifiers
imports Layout[C.1] Naturals[C.2]
exports
sorts Id
lexical syntax
[A-Z][A-Za-z0-9\-_. >\[\]*]* → Id
variables
ν [0 − 9 ′]* → Id

C.4 Identifier Containers

module Identifier-Containers
imports Booleans[A.2] Identifiers[C.3]
exports
sorts List[Id] Set[Id] Id×Id Set[Id×Id] Id×Set[Id] Id×List[Id]
Set[Id×Set[Id]] Set[Id×List[Id]]
context-free syntax

"[" {Id ","}* "]"	→ List[Id]	
"{" {Id ","}* "}"	→ Set[Id]	
"(" Id "," Id ")"	→ Id×Id	
"{" {Id×Id ","}* "}"	→ Set[Id×Id]	
"(" Id "," Set[Id] ")"	→ Id×Set[Id]	
"(" Id "," List[Id] ")"	→ Id×List[Id]	
"{" {Id×Set[Id] ","}* "}"	→ Set[Id×Set[Id]]	
"{" {Id×List[Id] ","}* "}"	→ Set[Id×List[Id]]	
Set[Id] "∩" Set[Id]	→ Set[Id]	{**left**}
Set[Id] "∪" Set[Id]	→ Set[Id]	{**left**}
Set[Id] "/" Set[Id]	→ Set[Id]	{**left**}
Id "∈" Set[Id]	→ Bool	
"(" Set[Id] ")"	→ Set[Id]	{**bracket**}
Id×Set[Id] "↪" Set[Id×Set[Id]]	→ Set[Id×Set[Id]]	
Set[Id×Id] "∪" Set[Id×Id]	→ Set[Id×Id]	{**left**}
sym "(" Set[Id×Id] ")"	→ Set[Id×Id]	

priorities
Set[Id] "∪"Set[Id] → Set[Id] < Set[Id] "∩"Set[Id] → Set[Id] <
Set[Id] "/"Set[Id] → Set[Id]
variables

$\mathcal{L}$ [0 − 9 ′]*	→ List[Id]
$\mathcal{S}^{\nu}$ [0 − 9 ′]*	→ Set[Id]
α [0 − 9 ′]*	→ Id×Id
"α*"[0 − 9 ′]*	→ {Id×Id ","}*
$\mathcal{A}$ [0 − 9 ′]*	→ Set[Id×Id]
θ [0 − 9 ′]*	→ Id×Set[Id]
ζ [0 − 9 ′]*	→ Id×List[Id]
$\mathcal{S}^{\theta}$ [0 − 9 ′]*	→ Set[Id×Set[Id]]
$\mathcal{S}^{\zeta}$ [0 − 9 ′]*	→ Set[Id×List[Id]]
"ν*"[0 − 9 ′]*	→ {Id ","}*

$\text{“}\nu^{+}\text{”}[0-9\,']* \to \{\textsc{Id}\ \text{“,”}\}+$
$\text{“}\theta^{*}\text{”}[0-9\,']* \to \{\textsc{Id}\times\textsc{Set}[\textsc{Id}]\ \text{“,”}\}*$
$\text{“}\zeta^{*}\text{”}[0-9\,']* \to \{\textsc{Id}\times\textsc{List}[\textsc{Id}]\ \text{“,”}\}*$

equations

Set of ν's

Remove the double entries:

[is-dbl] $\{\nu_1^*, \nu, \nu_2^*, \nu, \nu_3^*\} = \{\nu_1^*, \nu, \nu_2^*, \nu_3^*\}$

The usual set operations: “element of” ($\in$), union, intersection and difference:

[is-elt1] $\nu \in \{\} = \text{false}$
[is-elt2] $\nu \in \{\nu, \nu^*\} = \text{true}$
[is-elt3] $\nu \in \{\nu', \nu^*\} = \nu \in \{\nu^*\}$ **when** $\nu \neq \nu'$

[is-uni] $\{\nu_1^*\} \cup \{\nu_2^*\} = \{\nu_1^*, \nu_2^*\}$

[is-int1] $\{\} \cap \mathcal{S}^\nu = \{\}$
[is-int2] $\{\nu\} \cap \mathcal{S}^\nu = \{\}$ **when** $\nu \in \mathcal{S}^\nu = \text{false}$
[is-int3] $\{\nu\} \cap \{\nu_1^*, \nu, \nu_2^*\} = \{\nu\}$
[is-int4] $\{\nu_1^+, \nu_2^+\} \cap \mathcal{S}^\nu = \{\nu_1^+\} \cap \mathcal{S}^\nu \cup \{\nu_2^+\} \cap \mathcal{S}^\nu$

[is-dif1] $\{\} \;/\; \mathcal{S}^\nu = \{\}$
[is-dif2] $\{\nu\} \;/\; \mathcal{S}^\nu = \{\nu\}$ **when** $\nu \in \mathcal{S}^\nu = \text{false}$
[is-dif3] $\{\nu\} \;/\; \{\nu_1^*, \nu, \nu_2^*\} = \{\}$
[is-dif4] $\{\nu_1^+, \nu_2^+\} \;/\; \mathcal{S}^\nu = \{\nu_1^+\} \;/\; \mathcal{S}^\nu \cup \{\nu_2^+\} \;/\; \mathcal{S}^\nu$

Set of α's

Remove the double entries

[its-dbl] $\{\alpha_1^*, \alpha, \alpha_2^*, \alpha, \alpha_3^*\} = \{\alpha_1^*, \alpha, \alpha_2^*, \alpha_3^*\}$

Compute the union

[its-uni] $\{\alpha_1^*\} \cup \{\alpha_2^*\} = \{\alpha_1^*, \alpha_2^*\}$

Compute the “symmetric closure” of a set of α's

[its-sym0] $\text{sym}(\{\}) = \{\}$
[its-sym1] $\text{sym}(\{(\nu_1, \nu_2), \alpha^*\}) = \{(\nu_1, \nu_2), (\nu_2, \nu_1)\} \cup \text{sym}(\{\alpha^*\})$

Set of θ's

Remove the double entries:

[ists-dbl] $\{\theta_1^*, \theta, \theta_2^*, \theta, \theta_3^*\} = \{\theta_1^*, \theta, \theta_2^*, \theta_3^*\}$

Inject a θ definition in a set of θ's (unify the definitions if it was already in the set, add the definition otherwise).

[ists-inj1] $$(\nu_1, \mathcal{S}_1^\nu) \hookrightarrow \{\} = \{(\nu_1, \mathcal{S}_1^\nu)\}$$

[ists-inj2] $$(\nu, \mathcal{S}_1^\nu) \hookrightarrow \{(\nu, \mathcal{S}_2^\nu), \theta^*\} = \{(\nu, \mathcal{S}_1^\nu \cup \mathcal{S}_2^\nu), \theta^*\}$$

[ists-inj3] $$\frac{\nu_1 \neq \nu_2, \quad (\nu_1, \mathcal{S}_1^\nu) \hookrightarrow \{\theta_1^*\} = \{\theta_2^*\}}{(\nu_1, \mathcal{S}_1^\nu) \hookrightarrow \{(\nu_2, \mathcal{S}_2^\nu), \theta_1^*\} = \{(\nu_2, \mathcal{S}_2^\nu), \theta_2^*\}}$$

C.5 Dhal utilities

Finally, there are some functions on Dhal statements which can be used to extract information of those statements. The function '*bf*' checks whether a Dhal statement is in *basic form* (see definition 5.2). The function '*1*' checks whether a Dhal statement is not the composition of more statements. The function 'τ_σ' can be used to hide the statement σ in a series of statements (i.e. rename it to τ).

module Dhal-tools
imports Dhal-syntax$^{4.4.1}$ Booleans$^{A.2}$
exports
  context-free syntax
    bf "(" DHALSTAT ")" → BOOL
    1 "(" DHALSTAT ")" → BOOL
    τ "_" DHALSTAT "(" DHALSTAT ")" → DHALSTAT
equations

The *bf* function checks whether a statement is in basic form (i.e. contains no *goto* or *label* statements).

[bf-goto] $bf(\mathbf{goto}\ \nu) = \text{false}$
[bf-label] $bf(\mathbf{label}\ \nu) = \text{false}$
[bf-if] $bf(\mathbf{if}\ \mathcal{S}^\nu\ \mathbf{then}\{\sigma_1\}\ \mathbf{else}\{\sigma_2\}) = bf(\sigma_1) \wedge bf(\sigma_2)$
[bf-while] $bf(\mathbf{do}\{\sigma\}\ \mathbf{while}\ \mathcal{S}^\nu) = bf(\sigma)$
[bf-comp] $bf(\sigma_1; \sigma_2) = bf(\sigma_1) \wedge bf(\sigma_2)$
[bf-other] $bf(\sigma) = \text{true}$ **otherwise**

The *1* function checks whether a statement is not a composition of multiple statements.

[non-single] $1(\sigma_1; \sigma_2) = \text{false}$
[single] $1(\sigma) = \text{true}$ **otherwise**

The τ function removes a given statement from a (series of) statements.

[del-equal] $$\tau_\sigma(\sigma) = \tau$$

[del-if] $$\frac{\sigma \neq \mathbf{if}\ \mathcal{S}^\nu\ \mathbf{then}\{\sigma_1\}\ \mathbf{else}\{\sigma_2\}, \quad \tau_\sigma(\sigma_1) = \sigma_1',\ \tau_\sigma(\sigma_2) = \sigma_2'}{\tau_\sigma(\mathbf{if}\ \mathcal{S}^\nu\ \mathbf{then}\{\sigma_1\}\ \mathbf{else}\{\sigma_2\}) = \mathbf{if}\ \mathcal{S}^\nu\ \mathbf{then}\{\sigma_1'\}\ \mathbf{else}\{\sigma_2'\}}$$

[del-while]
$$\frac{\sigma \neq \mathbf{do}\{\sigma\}\ \mathbf{while}\ \mathcal{S}^{\nu},\quad \tau_{\sigma}(\sigma) = \sigma'}{\tau_{\sigma}(\mathbf{do}\{\sigma\}\ \mathbf{while}\ \mathcal{S}^{\nu}) \;=\; \mathbf{do}\{\sigma'\}\ \mathbf{while}\ \mathcal{S}^{\nu}}$$

[del-not-equal]
$$\tau_{\sigma_1}(\sigma_2) = \sigma_2 \quad \mathbf{when}\ \ \sigma_1 \neq \sigma_2 \quad \mathbf{otherwise}$$

[del-comp]
$$\frac{\tau_{\sigma}(\sigma_1) = \sigma_1',\ \ \tau_{\sigma}(\sigma_2) = \sigma_2'}{\tau_{\sigma}(\sigma_1;\, \sigma_2) \;=\; \sigma_1';\, \sigma_2'}$$

Appendix D

Variables used in the specification

$\mathcal{A}$	Set of tuples of two identifiers
α	Tuple of two identifiers
α^*	One or more tuples of two identifiers
b	Boolean
c	Character
c^+	One or more characters
δ	Declaration in a $\mathfrak{Dhal}$ program
δ^*	Zero or more declarations in a $\mathfrak{Dhal}$ program
ε	Continuation equation
ε^*	Zero or more continuation equations
$\mathcal{L}$	List of identifiers
ζ	Tuple of an identifier and a list of identifiers
ζ^*	Zero or more tuples of an identifier and a list of identifiers
ζ^+	One or more tuples of an identifier and a list of identifiers
n	Natural number
nc	Natural number constant
ν	Identifier
ν^*	Zero or more identifiers
ν^+	One or more identifiers
σ	Statement in a $\mathfrak{Dhal}$ program
$\mathcal{S}^\zeta$	Set of tuples of an identifier and a list of identifiers
$\mathcal{S}^\nu$	Set of identifiers
$\mathcal{S}^\theta$	Set of tuples of an identifier and a set of identifiers
θ	Tuple of an identifier and a set of identifiers
θ^*	Zero or more tuples of an identifier and a set of identifiers
θ^+	One or more tuples of an identifier and a set of identifiers

Note that identifiers (ν) are used for several purposes in the specifications presented in this thesis. They are used to denote variable access paths, nodes in graphs and the unknowns in a system of continuation equations.

Bibliography

[ADS91] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. Technical Report SERC-TR-93-P, Purdue University, Department of Computer Science, Software Engineering Research Center, July 1991.

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

[AH87] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1, pages 9–31. Ellis Horwood, 1987.

[AKPW83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the 10th ACM Annual Symposium on Principles of Programming Languages (POPL'83)*, pages 177–189, January 1983.

[AM72] E. Ashcroft and Z. Manna. The translation of 'goto' programs to 'while' programs. In *Proceedings of IFIP Congress 71*, pages 250–255, 1972.

[Amm90] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. Technical Report TR-1042, Center for Supercomputing Research and Development (CSRD), June 1990.

[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.

[Bak77] B. S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[Ban78] J. P. Banning. *A Method for Determining the Side Effects of Procedure Calls.* PhD thesis, Department of Electrical Engineering, Stanford, August 1978.

[Ban79] J. P. Banning. An efficient way to find the side effects of procedure calls and aliases of variables. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages (POPL'79)*, pages 29–41, 1979.

[BC92] M. Burke and J.-D. Choi. Precise and efficient integration of interprocedural alias information into data flow analysis. *ACM Letters on Programming Languages and Systems*, 1(1):14–21, March 1992.

[BHK89a] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BHK89b] J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[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.

[BKV96] M. G. J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation: an annotated bibliography. Technical Report P9603, University of Amsterdam, Programming Research Group, 1996.

[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.

[Chr75] N. Christofides. *Graph Theory – an Algorithmic Approach*. Academic Press, London, 1975.

[CK89] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 49–59, January 1989.

[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.

[Eas94] Easel Corporation, 25 Corporate Drive, Burlington, MA 01803. *Reference guide for the ESL language*, 1 edition, 1994.

[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 '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994. *SIGPLAN Notices,* 29(6).

[Eve79] S. Even. *Graph Algorithms.* Pitman Publ. Lmtd, London, 1979.

[FO76] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[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.

[Hen91] P. R. H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[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.

[Koo92] J. W. C. Koorn. GSE: A generic text and structure editor. In J. L. G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 168–177. SION, 1992. Appeared as Report P9202, University of Amsterdam.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[KV95] P. Klint and E. Visser. ToLaTeX: The program, 1995. See also [Vis95].

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN '88 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 '92 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 '93 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.

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

[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.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.

[Rey93] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, November 1993.

[RND77] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice Hall, New York, 1 edition, 1977.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 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.

[Vis95] E. Visser. ASF+SDF to LaTeX, 1995. Web site with links to the software and manuals. URL: `http://adam.wins.uva.nl/~visser/tolatex/`.

[Wal91] H. R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages (POPL'80)*, pages 83–94, January 1980.

[WL95] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. *SIGPLAN Notices*, 50(6).

[WO75] M. H. Williams and H. L. Ossler. Conversion of unstructured flow diagrams to structured. *The Computer Journal*, 21(2), 1975.