Growth-Rate Analysis of Flowchart Programs

Author: Aviad Pineles
Supervisor: Prof. Amir Ben-Amram

Project submitted in partial fulfillment of the requirements for an MSc degree in Computer Science, at the Academic College of Tel-Aviv Yaffo

August 2, 2014
1 Introduction

The field of static program analysis has its roots in the inception of computer science, and has a long and glorious tradition. The desire to be able to judge by a program’s structure, whether it ever reaches its end, and how long it takes to get there, has driven intense research in this field in the past, and present. Much of the research in complexity analysis in the past was done on input of programs in structured languages, including, in particular the previous work that constitutes the specific background to this project [5, 4]. These works left the application of their ideas to more low-level languages such as machine code and byte code, or more generally flowchart programs, as an open problem.

The crux of the problem of program analysis lies in the loop, which in structured languages is easy to detect, and therefore the focus can be put on its analysis. However, in flowcharts, loops are not explicit in the program source; in fact what constitutes a “loop” is not even well defined, making the fundamental task of identifying loops non-trivial at best. Current research has made some progress on the task of breaking flowcharts into loops, for example, by performing graph reduction. Graph reduction tries to recognize certain strongly connected subgraphs of the flowchart graph as loops. This is a classic topic in compilation and has been used in the context of complexity analysis by, e.g., [1, 2]. Another, more novel, direction of research is using ranking functions to detect loops [3, 6].

Another respect in which analysis of structured programs is different than that of flowcharts is that structured programs lend themselves to compositional analysis, as command semantics (as well as syntax) are compositional, and there is no non-local transfer of control (goto, etc.). In flowcharts, however, the compositional approach cannot work in general, which follows from the basic fact that a flowchart is not a recursive data type.

Our goal in this work is to be able to analyze a flowchart program and determine the output variables’ dependencies on the input. A dependency can be of several types: identity, linear, polynomial or exponential, and it represents the growth rate of the value of an output variable relative to the value of an input variable. The classic problem of determining the time of a program (as well as some other cost measures) is reducible to this growth-rate problem by introducing a virtual cost variable (in particular, step-count), incremented throughout the computation, and analyzing the growth-rate of this new variable.
In this article we describe a method of analyzing flowchart programs, while handling the two major issues stated above as follows. The first issue, detecting loops, is essentially bypassed by requiring the input to be *annotated*. This annotation partitions the arcs of the flowchart graph into loops, attaches a bounding variable to each loop, and arranges these loops in a *nesting tree*. This annotation must obey one rule, the bounding variable of a loop must not be modified in any arc belonging to that loop, or any loops nested below it in the nesting tree.

The second issue, analysis, we handle by analoging flowcharts and structured programs to automata and regular expressions respectively. Determining a regular expression that corresponds to a finite automaton (that is, represents the language that the automaton accepts) is a well known problem, with well known solutions, one of which is the *Rip* algorithm (also called the state elimination algorithm). We use a version of this algorithm to reduce the flowchart to a minimal form consisting of an entry node, an exit node, and one arc, while preserving the abstract semantics of the flowchart on this single arc. The particular abstract semantics we deal with in this work represents dependencies of output values on input values, as described above, and is based on [4].

The analysis of the flowchart in Figure 1, for example, produces the following results: Variables $g$, $i$, $j$ and $u$ grow at most polynomially as a function of the initial values of $b$, $e$ and $h$; Variables $c$ and $d$ grow at least exponentially as a function of the initial value of $b$. Firstly, note that the bound of the outer loop is $a$ which receives the value of $b$, and the bound of the inner loop is $e$, an input variable, which is reset after the first run of the inner loop. We can see that the polynomial growth is due to the fact that $g$, $i$, $j$ increase in the nested loop, by the value of variable $h$ which is unmodified in the program; and $u$ is (at the end) a copy of $g$. The number of iterations in the nested loop is polynomial in $b$ and $e$. Variables $c$ and $d$ are used to increment each other in the outer loop which runs a number of iterations bounded by $a$ and therefore by $b$, essentially doubling their values at most $b$ times.

One major advantage of being able to analyze flowcharts in this way stems from the flexibility of flowcharts. In a precursor to this work [4] the author describes a language called *core*, with a limited set of instructions dealing with integers (see Section 2.1), and then extends it to include the zero-assignment command (also known as a reset operation). Refer to Figure 4 and Figure 5 for examples of core programs and their annotated flowchart equivalents. In
the current work, we handle resets by reducing to the problem without resets. We describe an algorithm which ‘explodes’ the control-flow graph by creating copies of nodes based on their possible contexts (a context determines which variables hold zero), and modifying the instructions that connect them appropriately. This process creates a graph whose structure is altered, and in particular, loops can be disconnected, i.e., they are no longer part of a connected component. This kind of loop is handled in a straightforward way by our algorithms. This is shown in Figure 1 and Figure 2. Figure 1 shows a flowchart with two loops, the dashed arcs are the outer loop, and the dotted arcs are the inner loop (nested within the outer loop). Figure 2 shows the same flowchart after the “explosion” process, with the outer loop having been scattered into several components, which are not interconnected by dashed arcs. In other words, the outer loop is now not an SCC.

Figure 1: Sample flowchart – unprocessed representation. Dashed arcs represent the outer loop bounded by variable $a$, while dotted arcs represent the inner loop bounded by variable $e$. 
Figure 2: Sample flowchart – post-explosion representation. Dashed arcs represent the outer loop bounded by variable \( a \), while dotted arcs represent the inner loop bounded by variable \( e \). Note how the outer loop is not a strongly connected component.
We summarize the contributions of this work:

- We introduce loop-annotated flowcharts, a program form that eliminates the concern of deciding what subgraph of a flowchart should be considered “a loop.” This information can be obtained from different sources, but our work concentrates on analysis of such flowcharts. However, we show how such flowcharts can be easily obtained from structured programs; and also how a program transformation technique, informally called “CFG explosion,” directly benefits from the flexibility of our class of flowcharts.

- We present a technique by which a compositional analysis of a structured program can be transformed into an analysis of loop-annotated flowcharts. This technique is based on the conversion of automata to regular expressions. This technique may have interest beyond our specific application.

- By applying our technique to the analyses of [5, 4], we obtain the first algorithm that soundly and completely identifies polynomially-growing variables in our class of programs (a variant identifies linearly-bounded values). Thus, we extend the range of program types for which this problem is completely resolved, and bring the results of [5, 4] one step closer to “real” languages.

- Last but not least, our algorithms have all been implemented. Our demonstration program can process flowcharts in a textual input form or compile them from a structured program, apply all the algorithms described in this paper, and provide graphic visualization of the flowcharts. See Section 9 for a brief description of the software implementation.

2 Problem Definition

We define flowchart languages $FC_x$ where $x$ indicates the available set of atomic operations (assignments). Given a flowchart program $p \in FC_r$, the growth-rate problem is to determine the growth-rate of an output variable in relation to its input variables: specifically, whether it is polynomially bounded (a variant of our algorithm finds whether it is linear). The program’s
Variables hold nonnegative integers.

The syntax is described in Figure 3 and should be self-explanatory. In a command \( \text{loop } X \{ C \} \), variable \( X \) is not allowed to appear on the left-hand side of an assignment in the loop body \( C \). There is no special syntax for a “program.”

It is most convenient to assume that the only type of data is nonnegative integers. More generality is possible but will not be treated in this presentation.

The semantics of the core language is intended for over-approximating a realistic program’s semantics. Therefore, the core language is nondeterministic. The \textit{choose} command represents a nondeterministic choice, and can be used to abstract any concrete conditional command by simply ignoring the condition. The \textit{loop command} \( \text{loop } X \{ C \} \) repeats \( C \) a number of times bounded by the value of \( X \). Thus, it may be used to model different kinds of loops (for-loops, while-loops) as long as a bounding expression can be statically determined (possibly by an auxiliary analysis such as [7, 8]).
The use of bounded loops restricts the computable functions to be primitive recursive, but this is still rich enough to make the analysis problem challenging.

Formally, the semantics associates with every command $C$ over variables $x_1, \ldots, x_n$ a relation $⟦C⟧ ⊆ \mathbb{N}^n \times \mathbb{N}^n$. In the expression $\bar{x}[C]\bar{y}$, vector $\bar{x}$ (respectively $\bar{y}$) is the store before (after) the execution of $C$.

The semantics of skip is the identity. The semantics of an assignment leaves some room for variation: either the precise value of the expression is assigned, or a nonnegative integer bounded by that value. Because our analysis only involves monotone increasing functions, this choice does not affect the results; but the second, non-deterministic definition increases the expressiveness of the core language, since additional numeric operations can be modelled (over-approximated). Finally, composite commands are described by the equations:

$$\begin{align*}
⟦C_1; C_2⟧ &= ⟦C_2⟧ \circ ⟦C_1⟧ \\
⟦\text{choose } C_1 \text{ or } C_2⟧ &= ⟦C_1⟧ \cup ⟦C_2⟧ \\
⟦\text{loop } X_\ell \{C\}⟧ &= \{ (\bar{x}, \bar{y}) | \exists i : x_\ell \leq x_i \leq x_\ell \circ \cdot \cdot \cdot \circ C \} \\
\end{align*}$$

where $C]^i$ represents $C \circ \cdot \cdot \cdot \circ C$ ($i$ occurrences of $C$); and $C]^0 = ⟦\text{skip}⟧$.

For every command we also define its step count (informally referred to as running time). The step count of an atomic command is defined as 1. The step count of a loop command is the sum of the step counts of the iterations taken. Because of the nondeterminism in if and loop commands, the step count is also a relation.

### 2.2 The Flowchart Language

We define the language $FC_r$; we get $FC_{BJK}$ by omitting the reset instruction $x_i := 0$.

**Definition 2.1.** A program $p$ in $FC_r$ consists of:

- A finite set of variables $x_i$, $i \in \mathcal{I}$ (we usually use $n$ for the number of variables and $\mathcal{I} = \{1, \ldots, n\}$).

- A control-flow graph (CFG) which is a directed graph $G_p = (\text{Loc}_p, \text{Arc}_p)$. The nodes $\text{Loc}_p$ are known as locations.
• Each arc $a$ is labeled with an instruction, $\text{Inst}(a)$, of one of the instruction types listed below. We sometimes also refer to the arcs as “instructions;” the meaning will hopefully be clear from context.

• Essentially, the only type of instruction is assignment ($\text{skip}$ is considered a type of assignment). The assignment instructions are:

  1. $\text{skip}$, a no-op instruction.
  2. Variable assignment: $X_l := X_r$ ($\text{skip}$ may be considered a special case, with $l = r$).
  3. Constant multiple assignment: $X_l := k \times X_r$ where $k \geq 2$
  4. Addition: $X_l := X_r + X_s$
  5. Product: $X_l := X_r \times X_s$
  6. Reset: $X_l := 0$
  7. Unbounded value assignment: $X_l := \ast$

• There is a set of initial nodes (nodes with no predecessors), $P_{\text{init}}$, and a set of terminal nodes (nodes with no successors), $P_{\text{term}}$.

• A loop structure, refer to Definition 2.2 below.

Semantics: variables hold natural numbers (it is possible to extend the method to signed integers by considering their absolute value—but we would prefer to do that as preprocessing, reducing the problem to one dealing with natural numbers). Assignments are either interpreted verbatim, or as constraints (i.e., $X := Y + Z$ only asserts that the new value of $X$ is at most the sum of values of $Y$ and $Z$; it is common in program analysis to write such assertions in the form $x' \leq y + z$). As a consequence of the fact that we only analyze monotone operations, results are the same under these two interpretations\footnote{For variables other than the assigned one we will assume, under both interpretations, that they keep their values.}. This gives flexibility in using our programs as approximations of concrete programs in a richer language. Branching is non-deterministic. Information about looping in the (concrete) program is captured in the loop structure defined next. We define it formally first, explaining its significance afterwards.
Definition 2.2. A loop structure for program $p$ is a set $\mathcal{L}_p$ of disjoint subsets of $\text{Arc}_p$, called loops, which is endowed with the structure of a rooted tree, called the nesting tree. We write $L \preceq L'$ (or $L' \succeq L$) if $L$ is a descendant of $L'$ in this tree. We write $L$ for the union of $L$ and all its descendants. With each non-root $L \in \mathcal{L}_p$ is associated a variable, called its bound; formally $\text{Bound}(L) \in \mathcal{I}$. In addition, a “cut set” $\text{Cutset}(L) \subseteq L$ is defined. In a valid program, if $a \in L$, instruction $\text{Inst}(a)$ must not modify $X_{\text{Bound}(L')}$ for any $L' \succeq L$. In addition, a valid cut set has to satisfy the following: every cycle $C$ in the CFG includes an arc from the cutset of the lowest loop $L$ such that $L$ contains $C$ (called the smallest enclosing loop of $C$). Without loss of generality we shall assume that arcs in $\text{Cutset}(L)$ carry a special kind of annotation which we call $\text{ensureBound}$ or $\text{ensure}$ for short. An instruction annotated with $\text{ensure}$ implicitly includes a sort of decrement-and-test operation which ensures that flow passes through the arc at most $X_{\text{Bound}(L)}$ times.

Note that the notion of “a loop” is very flexible. It is a set of arcs, is not required to be strongly connected or even contain a cycle. In the most natural case—a flowchart created from a structured program—the nesting tree will correspond to the nesting of loops in the source program, we show this translation in section 3.2. The “bound” is intended to represent either a loop bound given explicitly in the source program, or one derived by some prior analysis (in the latter case an auxiliary variable would be created to store the bound). The root is intended to represent the part of the program outside all loops.

This flexible definition of a loop proves useful when dealing, for example, with flowcharts which are the result of optimized compiler output of a structured language. Compiler optimization might scatter parts of a loop body at different areas of the flowchart. Another example is when dealing with resets, our ‘explode’ algorithm might duplicate a loop’s arcs at various areas in the flowchart, depending on contexts.

Informally, the meaning of the loop bounds is this: once a loop is entered, computation can proceed through a cut-set arc (an $\text{ensure}$-annotated instruction) at most $n$ times before the loop is exited, where $n$ is the value of $X_L$. To exit means to execute an instruction outside the loop and its descendants.
2.3 The semantics, formally

Consider a program $p$ with variables $x_1, \ldots, x_n$, and control-flow graph $G_p = (\text{Loc}_p, \text{Arc}_p)$. For $a \in \text{Arc}_p$, write $a : P \to Q$ if $P$ is its source location and $Q$ its target.

**Definition 2.3** (states). A state of $p$ is $s = (P, \vec{x})$, where $P \in \text{Loc}_p$ is a program location and $\vec{x} \in \mathbb{N}^n$ represents an assignment of values to the variables.

**Definition 2.4** (transitions). A transition is a pair of states, a source state $s$ and a target state $s'$, related by an instruction $a$ of $p$. Specifically: $a : P \to Q$, $s = (P, \vec{x})$, $s' = (Q, \vec{x}')$, and the relation of $\vec{x}'$ to $\vec{x}$ is determined by the instruction $\text{Inst}(a)$ as explained earlier. When this holds, we write $s \xrightarrow{a} s'$.

**Definition 2.5** (trace). A trace of $p$ is a sequence of instructions that label a path in the CFG, which begins at an initial node.

**Definition 2.6** (transition sequence). A transition sequence of $p$ is a finite sequence of states $\tilde{s} = s_0, s_1, s_2, \ldots, s_t$ associated with a CFG path $a_1a_2\ldots a_t$ such that for all $i$, $s_{i-1} \xrightarrow{a_i} s_i$. We refer to the arcs $a_1, \ldots, a_t$ as the arcs of $\tilde{s}$.

Note that the definition of a transition sequence does not take the loop bounds into account. This is rectified in the next definition.

**Definition 2.7** (properly bounded). For a transition sequence $\tilde{s}$, let $L \in \mathcal{L}_p$ be the smallest loop (in the order $\preceq$) such that $L$ includes all arcs of $\tilde{s}$ (the smallest enclosing loop). Then, $\tilde{s}$ is properly bounded if the following conditions hold:

1. If $L$ is not the root, let $\ell = \text{Bound}(L)$; then the number of occurrences in $\tilde{s}$ of arcs of $\text{Cutset}(L)$ is at most the value of $x_\ell$ (which does not change throughout $\tilde{s}$).
2. If $L$ is the root, any $a \in L$ occurs at most once.
3. Every proper segment of $\tilde{s}$ is properly bounded.

**Example 2.8.** Consider a transition sequence corresponding to the path $bcabcabb$, where arc $a$ belongs to the root loop and arcs $b, c$ to a child loop $L$. Suppose further that $b$ is a cut-set arc. This sequence is proper if, and only if, the segments $bcbc$ and $bbb$ are proper; which requires the value of the bounding variable of loop $L$ to be at least 2 throughout the first segment, and
at least 3 throughout the second. Adding another a (where possible according to the graph) would make the transition sequence improper, because the root “loop” is not allowed to loop.

We say the properly-bounded transition sequence is a run of loop L when L is the smallest enclosing loop, as in the definition. If there is a run (of any loop) from state s to state s′, we write s ⇝ s′.

2.4 The Analysis Problem

Definition 2.9 (polynomial bound). The polynomial bound problem is: given p, determine, for each index j ∈ I, if there is a (multivariate) polynomial f_j(x̅) such that, whenever (p_i, x̅) ⇝ (p_t, x̅′), where p_i ∈ P_init and p_t ∈ P_term, it holds that x'_j ≤ f_j(x̅). If there is one, we say that output variable X_j is polynomially bounded.

Note that usually, the goal is to find a bound on the number of steps a program takes, rather than a dependency between input and output variables, however, the latter problem is reducible to the former by the introduction of a virtual ‘step count’ variable, even without the use of an increment instruction, by adding an additional unmodified dummy variable to the step count one at each program step, and determining the dependency between the two (see [?]).

3 Translation From Core to FC

We describe herein an algorithm for the translation of a program in the core language to our flowchart language.

3.1 Outline

The algorithm takes the abstract syntax tree of an input program, and builds a flowchart graph by inserting subgraphs corresponding to each grammar element.

The algorithm works by recursively traversing the input program AST, calling a Visit function at each node, which creates a single-entry single-exit flow control subgraph that represents the corresponding program component, and, in the case of loop commands, also adds a node to the loop tree.
At the end of the run, we obtain a flowchart that represents the entire input program, as well as the complete loop tree.

### 3.2 Formal Definition

1. Create a loop tree structure with just a root node, and initialize a tree pointer \textit{Current-Loop} that points to the root.

2. Call \textit{Visit(root)}, where \textit{root} is top level AST element representing the entire program. The function \textit{Visit(N)} is defined below for any grammar element \textit{N}.

\textbf{Definition 3.1.} \textit{Visit} is defined below by cases according to the node type. Any arcs created by this function call are also added to the arc set of the loop pointed to by the global variable \textit{Current-Loop}.

- **Command:** \textit{skip} – Return a CFG of the form:

  
  \[
  \begin{tikzpicture}
  \node (start) at (0,0) {start};
  \node (final) at (1,0) {Final};
  \node (skip) at (0.5,0) {skip};
  \draw[->] (start) -- (skip);
  \draw[->] (skip) -- (final);
  \end{tikzpicture}
  \]

- **Command:** \textit{x:=e} – Return a CFG of the form:

  
  \[
  \begin{tikzpicture}
  \node (start) at (0,0) {start};
  \node (final) at (1,0) {Final};
  \node (assign) at (0.5,0) {x := e};
  \draw[->] (start) -- (assign);
  \draw[->] (assign) -- (final);
  \end{tikzpicture}
  \]

  The instruction associated with the arc is taken verbatim from the source program. Unlike commands, expressions are not processed by this algorithm; \textit{Visit} is only called for commands.

- **Command:** \textit{C₁;C₂} – Let \( G₁ = Visit(C₁) \) and \( G₂ = Visit(C₂) \), return a CFG of the form:

  
  \[
  \begin{tikzpicture}
  \node (start) at (0,0) {start};
  \node (final) at (1,0) {Final};
  \node (G₁) at (0.5,0) {G₁};
  \node (G₂) at (1.5,0) {G₂};
  \draw[->] (start) -- (G₁);
  \draw[->] (G₁) -- (G₂);
  \draw[->] (G₂) -- (final);
  \draw[->] (start) -- (G₂);
  \end{tikzpicture}
  \]

  The initial state of the graph is the initial state of \( G₁ \) and the intermediate arc connects the final state of one graph to the initial state of the other.

- **Command:** \textit{choose C₁ or C₂} – Let \( G₁ = Visit(C₁) \) and \( G₂ = Visit(C₂) \), return a CFG of the form:
• **Command:** \texttt{loop \(X_l\) \{C\}} – Let \(G = Visit(C)\), create a loop tree node with \(X_l\) as its bound variable, with \texttt{Current-Loop} as its parent, and set \texttt{Current-Loop} to point at it. Return a CFG of the form:

Where \(X_l\) is the bound variable of the loop, and \texttt{ensure-skip} is a \texttt{skip} command annotated with \texttt{ensure} on that bound. After the recursive call to \(Visit(C)\) returns, set \texttt{Current-Loop} to point back at its parent.

3. As a final post-processing step we can make the graph more succinct by eliminating linear chains of transitions composed only of \texttt{skip} commands. Formally, any arc with a \texttt{skip} command which is outbound from a node with an out-degree of one (i.e. is the only arc outbound from that node) may be contracted. As an example, the following structure:

Will get converted to:
When performing this step care must be taken when dealing with the final and initial nodes of the flowchart. If the final node is merged, then the merged node becomes the new final node, and conversely if the initial node is merged, the merged node becomes the new initial node.

4 Regular Expressions

A core language program can be viewed as a special kind of regular expression. The alphabet is assignments and ensure commands. Note that although core doesn’t include explicit ensure annotations, we include them in the regular expression alphabet, and we use them for the analysis. However, it will be clear as the article develops, that such annotations are not necessary, and are actually altogether removed later on.

In this regular expression, command composition corresponds to concatenation, the choose command corresponds to alternation and the loop command corresponds roughly to the Kleene star operator applied to a grouping. However, in order to complete the correlation we must introduce another grouping construct which adds the information relevant to core loops. We denote this grouping construct with qualified square brackets \([\ldots \text{exp} \ldots]\). The semantics of this construct is that the total number of ensure-annotated commands matched by its content (according to regular expression matching) is bounded by the value of \(X_l\). For example the core language program presented in Code Snippet \([\text{I}]\) can be viewed as the regular expression \(A_{\text{copy},1,3}[5(EA_{\text{copy},2,3}A_{\text{sum},3,1,2})^*]A_{\text{copy},4,3}\). Where the \(A_{\ldots}\) symbols represent assignments, and the \(E\) symbol represents the ensure annotation, applied to the command immediately following it. Note that in general reverse translation is not possible, since there is no core language construct that corresponds, for example, to \(A_{\text{copy},2,1}(EA_{\text{sum},2,2})^*\), a loop that includes a ‘first iteration’ instruction. However, an annotated flowchart program is indeed equivalent to this special regular expression definition. By considering a flowchart program as an NFA, we will explain how to convert it to a suitable regular expression.

\(^2\)Concatenation, alternation, grouping and Kleene star as described in the definition of regular expressions. See [http://en.wikipedia.org/wiki/Regular_expression](http://en.wikipedia.org/wiki/Regular_expression)
with loop grouping operators.

By viewing regular expressions as generalizations of core language programs, as well as flowchart programs, we can adapt the analysis from [4] so that it applies to both kinds of programs.

4.1 Loop-Annotated Regular Expressions

First we begin by defining the syntax of our special kind of regular expression: the loop-annotated regular expression or *LARE*. A first attempt at a BNF definition of the regular expression grammar is presented as Code Snippet 2.
The symbols comprising the assignment and ensure commands include subscripts representing the type of command and its operands. For assignment commands, the first subscript indicates the type of the assignment, and the following 1, 2 or 3 subscripts represent variable indices which are the operands to the command. For example the symbol $A_{\text{sum},1,2,3}$ represents the assignment $X_1 := X_2 + X_3$.

To this first attempt we must add that any recurrence (‘*’) must occur within a loop grouping (‘ [ . . . ] ’) for the expression to be valid in LARE. To express this in the BNF grammar requires a simple but cumbersome modification to some definitions as presented in Code Snippet 3.
We must furthermore add to that definition that every recurrence term must contain ensure-annotated commands (one or more), such that every string matched by it will contain one. This is harder to express in BNF so we will leave this validity check for later in the parsing process. Note, however, that because of the aforementioned restriction on the ensure annotation we may state that the annotation must occur at the beginning of each recurrence term, and that it can be implicit as such. I.e. an ensure annotation is implicit on the first assignment after every opening parenthesis (‘(‘). We
shall denote this variation of the language by $LARE'$ with a BNF definition similar to that in Code Snippet 3 excluding the `<ensure>` term.

4.2 Translation from $LARE$ to $LARE'$

It is easy to translate a $LARE$ program to a $LARE'$ one by simply removing all ensure symbols (‘E’). This is correct because a valid $LARE$ program includes an ensure annotation inside every recurrence term and outside of any alternation term (or present in all its constituents). Also, multiple ensure annotations directly inside the same concatenation term (i.e. not counting those in nested terms) can be replaced by a single one. The reason for this is that these commands occur as an atomic group in the program, i.e., whenever one is executed, the rest are also executed. This means that replacing them with a single command is only going to change the number of iterations subject to the loop bound by a constant, which will not affect the growth-rate properties of the program.

For example, the $LARE$ program $ab(cEd(fEg)\ast E)\ast$ is equivalent to the following $LARE'$ program $ab(cd(fg)\ast)\ast$. Where lower case letters represent assignment statements, and a capital E represents the ensure symbol.

4.3 $LARE \leftrightarrow FC$ Equivalence

Disregarding the loop structure, a flowchart is an arc-labeled graph with distinguished initial and final nodes; in other words, an NFA. It is therefore equivalent (in terms of the traces it describes) to a regular expression. The $LARE$ language adds the capability of expressing the loop structure. It is designed just so that an FC program can be converted to a $LARE$ program. This is done by post-order traversal of the loop tree, generating regular expressions for the loops, and enclosing each such regular expression by the appropriate loop grouping.

Our algorithm is based on the classical Rip algorithm ([10], pp. 63–69]), which transforms an NFA into a regular expression. We use the part which rips a single node without any essential modification, but change the enclosing steps in order to add loop information to the resulting expression.
4.3.1 The Conversion Algorithm

First we define an algorithm $Rip$ which converts an NFA represented as a directed multigraph, with regular expressions on the arcs, into a bipartite graph with the partite sets of its initial and final states, and arcs labeled with regular expressions, such that for every initial node $i$ and final node $f$, the regular expression connecting $i$ to $f$ in the resulting graph describes the set of all traces which begin at $i$ and end at $f$.

$Rip$ uses a procedure, denoted $RipOne$ for removing one state from the NFA while keeping the language it accepts without change. We shall begin by defining the algorithm for $RipOne$, which accepts a graph $g$ and a node $v$ to be ripped:

**Definitions:**

- $v$ denotes the state to be removed from the graph $g$.
- Denote by $Rex_e$ the regular expression on the arc $e$.

$RipOne(g,v)$

1. Replace all self loops of $v$ with a single loop labeled with the alternation of the regular expressions on the original self loops. Denote this self loop by $Rex_{vv}$.

2. For every path of length two $e_1e_2$ with $v$ as its middle node replace the path with an arc $e'$ with the following regular expression:
   - If $v$ has a self loop -
     \[ Rex_{e'} = Rex_{e_1}(Rex_{vv})^*Rex_{e_2} \]
   - Otherwise -
     \[ Rex_{e'} = Rex_{e_1}Rex_{e_2} \]

3. Remove $v$ from the graph.

At this point our graph might be a multigraph, which we will convert to a simple graph:
4. For every pair of nodes \((s,t)\) which are connected by more than one arc, replace all such arcs with a single arc with the regular expression which is the alternation of the regular expressions on the original arcs.

Next we define the Rip procedure, which accepts a graph \(g\) and a bound variable represented by an integer, denoted by \(l\). We assume that all initial states of \(v\) have no inbound arcs, and all final states of \(v\) have no outbound arcs. If that is not the case, introduce new initial and final nodes appropriately for every such node.

\[
\text{Rip}(g, l)
\]

- For every node \(v\) which is not an initial or final node in \(g\), perform \(\text{RipOne}(g, v)\).
- Do this iteratively until no more nodes remain which are not initial or final ones. The resulting graph is a bipartite graph between the initial nodes and the final ones.
- For every arc \(e\) in this bipartite graph, replace its regular expression with a bracketed one:

\[
\text{Rex}_e := [l \text{Rex}_e]
\]

Finally we describe the conversion algorithm itself. The algorithm which we shall call ConvertFC will accept an annotated flowchart as input (i.e. a graph and a nesting tree of loops). Since the loop tree, denoted by \(L\), contains all the information in the annotated flowchart, we will use it as the input. Every node \(L\) in the tree represents the set of arcs belonging to the loop \(L\), and its bounding variable is denoted by \(x_{\text{Bound}(L)}\). We denote by \(\overline{L}\) the set of arcs belonging to the loop \(L\) and all its descendants.

Definitions

1. **Outer boundary node** - An outer boundary node of \(L\) is a node which has an incident arc in \(L\) and an incident arc not in \(\overline{L}\).

2. **Inner boundary node** - An inner boundary node of \(L\) is a node which has an incident arc in \(L\) and an incident arc in \(\overline{L} \setminus L\).
3. **Internal node** - An *internal node* of $L$ is a node which has an incident arc in $L$, and is neither an inner boundary node of $L$, nor an outer boundary node of $L$.

4. **Virtual input node** - A *virtual input node* of an outer boundary node $v$, denoted by $v_{in}$, is a kind of node that we create temporarily in the algorithm, it has all the incident arcs of $v$ except inbound arcs in $L$.

5. **Virtual output node** - A *virtual output node* of an outer boundary node $v$, denoted by $v_{out}$, is a node (created during the algorithm) which has all the incident arcs of $v$ except for outbound arcs in $L$.

The role of boundary nodes is to provide a standardized ‘socket’ for connecting the graph returned by the recursive call, to the graph at the call site. These ‘sockets’ are created in the post-order execution of the algorithm, at the leaf nodes first, and then used by their ancestors.

When processing a loop $L$, the algorithm described below returns a directed acyclic bipartite graph connecting virtual input nodes and virtual output nodes.

**ConvertFC($L$)**

1. Perform recursive calls to $ConvertFC$ for all the children of $L$. The result is a possibly disconnected collection of bipartite graphs (graphs resulting from the processing of different subloops may share nodes, because the subloops may share nodes in the FC).

2. Connect inner boundary nodes of $L$ to corresponding virtual nodes returned by the recursive calls. Specifically, for every inner boundary node $v$, create arcs $(v, v_{in})$ and $(v_{out}, v)$ with $\epsilon$ transitions.

3. Let $g'$ be graph consisting of the union of $L$, graphs returned by the recursive calls and arcs added in the previous step.

4. For every virtual node $v_{in}$ or $v_{out}$ in $g'$ perform $RipOne(g', v)$, leaving the resulting graph with no virtual nodes.

5. For every outer boundary node $v$ of $L$, create virtual input and output nodes $v_{in}$ and $v_{out}$. Create arcs $(v_{in}, v)$ and $(v, v_{out})$ with $\epsilon$ transitions. Note that these virtual boundary nodes are returned to the caller (and used above in Step 2).
6. Perform $Rip(g', x_{\text{Bound}(L)})$ and return the result.

### 4.3.2 ConvertFC Example

Consider the following flowchart:

![Flowchart](chart.png)

With the following loop tree:

- $L_0 = \{a, b, e\}$ - Root loop
- $L_1 = \{c, d\}$ - $\text{Bound}(L_1) = 1$

We start by processing the arcs of loop $L_1$. The graph induced by the arcs of $L_1$ is:

![Induced Graph](graph.png)

This graph has no child loops, therefore it has no inner boundary nodes. However, both $B$ and $C$ in this graph are outer boundary nodes because of arcs $b$ and $e$ respectively. Therefore we transform this graph as follows:
Next we perform Rip on this graph which rips out nodes $B$ and $C$ and keeps all the rest.

Next we process loop $L_0$. This loop has no outer boundary nodes and two inner boundary nodes: $B$ and $C$. The algorithm first creates the arcs connecting the inner boundary nodes to the virtual nodes returned by the recursive calls (the dashed nodes are the virtual nodes associated with the inner boundary nodes $B$ and $C$. This is the so-called ‘socket’ mentioned above):
Then we attach the inner graph, according to the algorithm:
Note that this graph has two sources: $I$ and $C_{in}$, and two sinks: $F$ and $B_{out}$, however we don’t care about the regular expressions resulting from starting at $C_{in}$ or ending at $B_{out}$. Also note that there are multiple arcs connecting $B_{in}$ and $B_{out}$ as well as $C_{in}$ and $C_{out}$, but in this case the result of merging them is as if the $\epsilon$ transition didn’t exist. Next we perform $Rip$ on this graph to get:

The end result we are interested in is the regular expression between $I$ and $F$, which is

$$[0 A_{skip} A_{copy,1,2} [1 E (A_{sum,3,5,2}E)^*] A_{copy,4,2}]$$
5  Interim Summary

We have so far referred to several languages:

1. Core\textsubscript{BJK} - A structured language as defined in \cite{5}
2. Core\textsubscript{r} - Core\textsubscript{BJK} with the addition of the reset assignment, as in\cite{4}.
3. FC\textsubscript{bjk} - Annotated flowchart with statement types according to BJK (i.e. no resets), with some extensions: Multiplication by a constant, and unbounding (\(X:=**\)).
4. FC\textsubscript{r} - FC\textsubscript{bjk} with the addition of the reset assignment.
5. LARE - Loop-annotated regular expressions with an explicit \texttt{ensure} symbol.
6. LARE' - Loop-annotated regular expressions with the \texttt{ensure} symbol implicit in the opening parenthesis of recurrence terms.

We have described the following translations:

1. Core (any variant) to LARE
2. Core (any variant) to FC (the corresponding variant)
3. FC (any variant) to LARE (and vice versa)
4. LARE to LARE' (and vice versa - the reverse translation is trivial).

6  Analysis of LARE' programs

We define an interpretation of LARE' programs over the domain of dependency sets. By applying this interpretation, we shall find how the magnitude of variable values at the end of the computation of the given program depends on the initial values.
6.1 Dependencies and dependency sets

Definition 6.1. The set of *dependency types* is \( \mathbb{D} = \{1, 1^+, 2, 3\} \), with order \( 1 < 1^+ < 2 < 3 \), and binary maximum operator \( \sqcup \). We write \( x \simeq 1 \) for \( x \in \{1, 1^+\} \).

Verbally, we may refer to these types as:

- \( 1 = \text{identity dependency} \)
- \( 1^+ = \text{additive dependency} \)
- \( 2 = \text{multiplicative dependency} \)
- \( 3 = \text{exponential dependency} \)

Definition 6.2. The set of *dependencies* is \( \mathbb{F} \), which is the union of two sets:

1. The set of unary dependencies, isomorphic to \( \mathcal{I} \times \mathbb{D} \times \mathcal{I} \). The notation for an element is \( i \delta \rightarrow j \).
2. The set of binary dependencies, isomorphic to \( \mathcal{I} \times \mathcal{I} \times \mathcal{I} \times \mathcal{I} \), where the notation for an element is \( i j \Rightarrow k \ell \).

Definition 6.3. A *dependency set* is a subset of \( \mathbb{F} \), subject to the constraint that a binary dependency \( i j \Rightarrow k \ell \) may appear in the set only when it also includes \( i \alpha \rightarrow k \in S \) and \( j \beta \rightarrow \ell \in S \), for \( \alpha, \beta \simeq 1 \), and \( i \neq j \lor k \neq \ell \).

Definition 6.4. The *identity dependency set* is

\[
ID_{\text{dep}} \overset{\text{def}}{=} \text{COMPLETE}( \{ i \downarrow \rightarrow i \mid i \in \mathcal{I} \} )
\]

Definition 6.5. The function \( \text{COMPLETE} \) adds to a dependency set all binary dependencies that can be added according to the above rule. That is:

\[
\text{COMPLETE}(S) = S \cup \{ i j \Rightarrow \ell \mid i \alpha \rightarrow k \in S \mathbin{\land} j \beta \rightarrow \ell \in S
\]
\[
\text{for some } \alpha, \beta \simeq 1 \}.
\]

6.2 Interpretation of LARE’ operations

To give a dependency-set semantics to an LARE’ program \( \mathbf{e} \), which we denote by \( \llbracket \mathbf{e} \rrbracket_{\text{dep}} \), we give a semantics to every symbol and and every operation.
6.2.1 Symbols

The symbols represent assignment instructions, and associating dependency sets to them is quite intuitive. Thus

\[
\begin{align*}
\llbracket A_{\text{skip}} \rrbracket_{\text{dep}} &= I \cdot D_{\text{dep}} \\
\llbracket A_{\text{unbound}, r} \rrbracket_{\text{dep}} &= \text{COMPLETE}( \{ i \to i \mid i \neq r \} ) \\
\llbracket A_{\text{copy}, r, s} \rrbracket_{\text{dep}} &= \text{COMPLETE}( \{ s \to r \} \cup \{ i \to i \mid i \neq r \} ) \\
\llbracket A_{\text{sum}, r, s, t} \rrbracket_{\text{dep}} &= \text{COMPLETE}( \{ s^{1+} \to r, t^{1+} \to r \} \cup \{ i \to i \mid i \neq r \} ) \text{ when } s \neq t \\
\llbracket A_{\text{sum}, r, s, s} \rrbracket_{\text{dep}} &= \text{COMPLETE}( \{ s^{2} \to r \} \cup \{ i \to i \mid i \neq r \})
\end{align*}
\]

Note that it is possible to extend the set of symbols, and in fact invent a symbol for any dependency set that one wants to use as the interpretation of some command or basic block in the subject programming language. This paves the way for using our analysis algorithms for richer languages.

6.2.2 Operators

We explain how to interpret the LARE operators: concatenation, alternation and recurrence.

**Alternation** is interpreted by set union: for \( E, F \) in the syntactic class \(<re>\) or \(<re’>\),

\[
\llbracket EF \rrbracket_{\text{dep}} = \llbracket E \rrbracket_{\text{dep}} \cup \llbracket F \rrbracket_{\text{dep}}
\]

**Concatenation** is interpreted by composition: for \( E, F \) in the syntactic class \(<re>\) or \(<re’>\),

\[
\llbracket EF \rrbracket_{\text{dep}} = \llbracket E \rrbracket_{\text{dep}} \cdot \llbracket F \rrbracket_{\text{dep}}
\]

where the last operation is, naturally, the component-wise product of \( E \) and \( F \), so it requires us to define the product of dependencies: and this is given by

31
Definition 6.6 (dependency composition). The binary operation $\cdot$ is defined on $F$ by the following rules:

\[
(i \overset{\alpha}{\to} j) \cdot (j \overset{\beta}{\to} k) = i \overset{\alpha \cup \beta}{\to} k
\]

\[
(i \overset{\alpha}{\to} j) \cdot (j \overset{\gamma}{\Rightarrow} k') = \begin{cases} 
(i \overset{\gamma}{\Rightarrow} k'), & \text{if } \alpha \not\simeq 1 \\
(i \overset{\alpha}{\to} j) \cdot (j \overset{\gamma}{\Rightarrow} k), & \text{if } \alpha \simeq 1
\end{cases}
\]

\[
(i \overset{\alpha}{\Rightarrow} j) \cdot (j \overset{\alpha}{\Rightarrow} k') = \begin{cases} 
(i \overset{\alpha}{\Rightarrow} k'), & \text{if } i \neq i' \text{ or } k \neq k' \\
(i \overset{\alpha}{\Rightarrow} k), & \text{if } i = i' \text{ and } k = k'
\end{cases}
\]

Recurrence, or the star operator, is (as expected) the most complex. Its handling involves computation of a closure under composition, which we denote by $LFP$; more precisely, $LFP(S)$ is defined as the least fixpoint (under set containment) of the function:

\[
f(X) = ID_{dep} \cup X \cup (X \cdot S).
\]

It also involves the loop correction operator, defined on single dependencies by

\[
LC_\ell(1^+ i) = \ell \overset{2}{\Rightarrow} i
\]

\[
LC_\ell(2^+ i) = \ell \overset{3}{\Rightarrow} i
\]

\[
LC_\ell(\Delta) = \Delta \quad \text{for all other } \Delta \in \mathbb{F}.
\]

Using these definitions,

\[
[E^*]_{dep} = S \cdot LC_\ell(S) \cdot S,
\]

where $S = LFP([E]_{dep})$ and $\ell$ is the index of the bounding variable for the closest enclosing bracket construct.

**Example 6.7** (abstract evaluation). The abstract evaluation of $A_{\text{copy},1,3}[A_{\text{copy},2,3}A_{\text{sum},3,1,2}]^*A_{\text{copy},4,3}$, which corresponds roughly to Code Snippet[1] except for the unnatural placement of the closing bracket, which has been done just for the sake of the example.
\[
[A_{copy,2,3}]_{dep} = \text{COMPLETE}( \{3 \mapsto 2, 1 \mapsto 1, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5 \} )
\]
\[
= \{ 1 \xrightarrow{1} 1, 3 \xrightarrow{1} 2, 3 \xrightarrow{1} 3, 4 \xrightarrow{1} 4, 5 \xrightarrow{1} 5, 1 \xrightarrow{1} 1 \mapsto 1, 3 \xrightarrow{1} 3 \mapsto 3, 1 \xrightarrow{1} 1 \mapsto 1, 4 \xrightarrow{1} 4 \mapsto 4 \}
\]

\[
[A_{sum,3,1,2}]_{dep} = \text{COMPLETE}( \{1 \mapsto 3, 2 \mapsto 3, 1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 4, 5 \mapsto 5 \} )
\]
\[
= \{ 1 \xrightarrow{1} 1, 1 \xrightarrow{1} 3, 2 \xrightarrow{1} 2, 2 \xrightarrow{1} 3, 4 \xrightarrow{1} 4, 5 \xrightarrow{1} 5, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 3 \xrightarrow{1} 3 \mapsto 3, 1 \xrightarrow{1} 1 \mapsto 1, 2 \xrightarrow{1} 2 \mapsto 2, 5 \xrightarrow{1} 5 \mapsto 5 \}
\]

The result of the composition \([A_{copy,2,3}A_{sum,3,1,2}]_{dep}\) is the set of all \(a \cdot b\) where \((a, b) \in [A_{copy,2,3}]_{dep} \times [A_{sum,3,1,2}]_{dep}\) and \(a \cdot b\) is defined:

\[
= \{ 1 \xrightarrow{1} 1, 1 \xrightarrow{1} 3, 3 \xrightarrow{1} 2, 2 \xrightarrow{1} 3, 4 \xrightarrow{1} 4, 5 \xrightarrow{1} 5, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 3 \xrightarrow{1} 3 \mapsto 3, 1 \xrightarrow{1} 1 \mapsto 1, 2 \xrightarrow{1} 2 \mapsto 2, 5 \xrightarrow{1} 5 \mapsto 5 \}
\]

Let \(E_1 = A_{copy,2,3}A_{sum,3,1,2}\), we need to compute the dependencies of recurrence \([E_1^*]_{dep}\). To show the calculation of the fixpoint we will start at the value of \(ID_{dep}\) and iteratively compose it with the value of \([E_1]_{dep}\) until we reach the fixpoint.

\[
ID_{dep} = \{ 1 \xrightarrow{1} 1, 2 \xrightarrow{1} 3, 3 \xrightarrow{1} 4 \xrightarrow{1} 4, 5 \xrightarrow{1} 5, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 3 \xrightarrow{1} 3 \mapsto 3, 1 \xrightarrow{1} 1 \mapsto 1, 2 \xrightarrow{1} 2 \mapsto 2, 5 \xrightarrow{1} 5 \mapsto 5 \}
\]

Iteration no. 1:

\[
\{ 1 \xrightarrow{1} 1, 1 \xrightarrow{1} 3, 2 \xrightarrow{1} 2, 3 \xrightarrow{1} 3, 3 \xrightarrow{1} 3, 4 \xrightarrow{1} 4, 5 \xrightarrow{1} 5, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 3 \xrightarrow{1} 3 \mapsto 3, 1 \xrightarrow{1} 1 \mapsto 1, 2 \xrightarrow{1} 2 \mapsto 2, 5 \xrightarrow{1} 5 \mapsto 5 \}
\]

Iteration no. 2:

\[
\{ 1 \xrightarrow{1} 1, 1 \xrightarrow{1} 3, 3 \xrightarrow{1} 3, 2 \xrightarrow{1} 2, 3 \xrightarrow{1} 2, 3 \xrightarrow{1} 3, 3 \xrightarrow{1} 3, 4 \xrightarrow{1} 4, 5 \xrightarrow{1} 5, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 1 \xrightarrow{1} 1 \mapsto 1, 3 \xrightarrow{1} 3 \mapsto 3, 1 \xrightarrow{1} 1 \mapsto 1, 2 \xrightarrow{1} 2 \mapsto 2, 5 \xrightarrow{1} 5 \mapsto 5 \}
\]
Iteration no. 3:

\[ \{ 1 \mapsto 1, 1 \mapsto 2, 1 \mapsto 3, 1 \mapsto 2, 2 \mapsto 3, 2 \mapsto 3, 3 \mapsto 3, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 5 \mapsto 5 \} \]

And this is the fixpoint. Next we apply loop-correction with bounding variable 5 to the result:

\[ \{ 1 \mapsto 1, 1 \mapsto 2, 1 \mapsto 3, 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 2, 3 \mapsto 3, 4 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 5 \mapsto 5 \} \]

Next we compute \( S \cdot LC(S) \cdot S \), where \( S \) is the fixpoint dependency set:

\[ \{ 1 \mapsto 1, 1 \mapsto 2, 1 \mapsto 3, 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 5 \mapsto 5 \} \]

Next step:

\[ [A_{copy,4,3}]_{dep} = \text{COMPLETE}( \{ 3 \mapsto 4, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 5 \mapsto 5 \} ) \]

\[ = \{ 1 \mapsto 1, 1 \mapsto 2, 1 \mapsto 3, 1 \mapsto 4, 5 \mapsto 5, 2 \mapsto 1, 3 \mapsto 3 \mapsto 3 \mapsto 4, 5 \mapsto 5, 5 \mapsto 5 \} \]

\[ [E_{1}A_{copy,4,3}]_{dep} = \]

\[ \{ 1 \mapsto 1, 1 \mapsto 2, 1 \mapsto 3, 1 \mapsto 4, 4 \mapsto 2, 2 \mapsto 3, 2 \mapsto 3, 3 \mapsto 3, 3 \mapsto 4, 3 \mapsto 3, 3 \mapsto 4, 3 \mapsto 4, 4 \mapsto 2, 4 \mapsto 3, 5 \mapsto 3 \} \]
Just two more steps, first compute:

\[
\left[ A_{copy,1,3} \right]_{dep} = \text{COMPLETE}( \{ 3 \rightarrow 1, 2 \rightarrow 3, 3 \rightarrow 1, 4 \rightarrow 5, 5 \rightarrow 5 \} )
\]

= \{ 2 \rightarrow 1, 2 \rightarrow 3, 1 \rightarrow 3, 4 \rightarrow 5, 1 \rightarrow 5, 2 \rightarrow 1, 2 \rightarrow 2, 2 \rightarrow 5, 2 \rightarrow 2, 2 \rightarrow 2, 2 \rightarrow 5, 2 \rightarrow 5, 3 \rightarrow 3, 4 \rightarrow 4, 5 \rightarrow 5, 3 \rightarrow 3, 4 \rightarrow 4, 3 \rightarrow 5, 5 \rightarrow 5, 5 \rightarrow 5 \}

And then compose it with the previous step. This composition yields unary and binary dependencies (as usual), however since it is the output of our algorithm, we are only interested in the unary dependencies, which are:

\[
= \{ 2 \rightarrow 1, 2 \rightarrow 3, 1 \rightarrow 3, 4 \rightarrow 5, 1 \rightarrow 5, 2 \rightarrow 1, 2 \rightarrow 2, 2 \rightarrow 5, 2 \rightarrow 2, 2 \rightarrow 2, 2 \rightarrow 5, 2 \rightarrow 5, 3 \rightarrow 3, 4 \rightarrow 4, 5 \rightarrow 5, 3 \rightarrow 3, 4 \rightarrow 4, 3 \rightarrow 5, 5 \rightarrow 5, 5 \rightarrow 5 \}
\]

Note that there are multiple dependencies of different bounds between the same variables, for example \(3 \rightarrow 2, 3 \rightarrow 2\) and \(3 \rightarrow 2\). In these cases we can safely keep only the dependency of the highest bound, i.e., in this case, we keep \(3 \rightarrow 2\). The above dependency set when filtered in this way contains:

\[
= \{ 2 \rightarrow 1, 2 \rightarrow 3, 1 \rightarrow 3, 4 \rightarrow 5, 1 \rightarrow 5, 2 \rightarrow 1, 2 \rightarrow 2, 2 \rightarrow 5, 2 \rightarrow 2, 2 \rightarrow 5, 2 \rightarrow 5, 3 \rightarrow 3, 4 \rightarrow 4, 5 \rightarrow 5, 3 \rightarrow 3, 4 \rightarrow 4, 3 \rightarrow 5, 5 \rightarrow 5, 5 \rightarrow 5 \}
\]

And in plain english:

1. Variable 1 is bounded by variable 3.

2. Variable 2 is unmodified in some traces, and is bounded at most polynomially by variables 3 and 5 in others.

3. Variables 3 and 4 are bounded at most polynomially by variables 3 and 5.

4. Variable 5 is not modified in the program.
7 Analysis Algorithm for FC Programs

7.1 Naïve Version

The principle of the algorithm is first converting the source program to a LARE’ program, and then performing the analysis on the result. This is inefficient because converting an NFA to a regular expression is not polynomially bounded.

7.2 Function Fusion

To formulate a more efficient algorithm we will use a principle called function fusion (see [11]). Using function fusion we do not split the analysis into two distinct steps – conversion from FC to LARE’, and then analysis of the LARE’ result. Rather, we work directly on a graph with dependency sets as arc labels, and perform the appropriate operations on the resulting sets to reach the end result, without generating and discarding regular expressions on the way.

We adapt the algorithm for conversion from FC to LARE presented in 4.3.1 to produce dependencies directly, without going through the intermediate regular expression phase.

Ingredients The basic ingredient in this algorithm is called a dependence fact. In its simple (unary) form, it indicates that an output variable depends, in a certain way, on some input variable. The set of variable indices is denoted by $I$ with generic elements $i, j, k$ etc.

Definition 7.1 (dependency set composition). The binary operation $\cdot$ is defined on a dependency set $D$ as the set of all defined compositions $a \cdot b$ where $(a, b) \in D \times D$.

Our input is an NFA represented as a directed multigraph, with dependency sets on the arcs, and we transform it into a bipartite graph between its initial and final states. This bipartite graph will have its arcs labeled with the dependency sets appropriate for the specific initial and final nodes incident to them. For an arc $e$ we denote its label by $Dep_e$.

Definition 7.2 (dependency set fixpoint). We define the function $LFP(d)$ which returns the least fixpoint of the composition operator on the depen-
dency set $d$. More precisely, $LFP(d)$ is defined as the fixpoint of the function:

$$f(D) = D \cdot d \cup D \cup \text{Skip}$$

We shall begin by presenting the adaptation of the $RipOne$ algorithm which accepts a graph $g$ and a node $v$ to be ripped, similarly to the regular expression one; and receives an extra argument $l$ which is the bound variable of the currently processed loop:

$\text{RipOne}(g, v, l)$

1. Denote by $vv_i$ the $i^{th}$ self loop connecting $v$ to itself, and $Dep_{vv} = \bigcup_i Dep_{vv_i}$. Replace all self loops $vv_i$ of $v$ with a single loop labeled with the dependency set $LC_l(LFP(Dep_{vv}))$. Denote this self loop by $vv$. For generality if $v$ has no self loops, $Dep_{vv} = \emptyset$

2. For every path of length two $e_1e_2$ with $v$ as its middle node replace the path with an arc $e'$ with the following dependency set:
   - If $e_1e_2$ is a cycle (i.e., $e'$ will be a self loop) - $Dep_{e'} = LC_l(LFP(Dep_{e_1} \cdot Dep_{vv} \cdot Dep_{e_2}))$
   - Otherwise - $Dep_{e'} = Dep_{e_1} \cdot Dep_{vv} \cdot Dep_{e_2}$

3. Remove $v$ from the graph.

At this point our graph might be a multigraph, which we will convert to a simple graph:

4. For every pair of nodes $(s, t)$ which are connected by more than one arc, replace all such arcs with a single arc with the dependency set which is the union of the dependency sets on the original arcs.

Next we adapt the $Rip$ procedure:

$\text{Rip}(g, l)$

1. For every node $v$ which is not an initial or final node in $g$, perform $\text{RipOne}(g, v, l)$. 

37
2. Do this iteratively until no more nodes remain which are not initial or final ones. The resulting graph is a bipartite graph between the initial nodes and the final ones.

Finally we adapt the conversion algorithm ConvertFC, which is almost identical to the one described above in Section 4.3 except that dependency sets are used instead of regular expressions, and empty dependency sets are used wherever \( \epsilon \)-transitions were used in the original algorithm.

### 7.3 Execution Traces

This algorithm also supports execution traces, i.e., The ability to reconstruct a trace that results in a particular dependency output by the algorithm. During initialization, when commands are first transformed into dependency sets (see Section 6.2.1 for a LARE example, but the case for FC\(_r\) commands is similar), the arc of the command is associated with each dependency in that set, this is the trace of the dependency. Later on during the algorithm, whenever new dependencies are created, for example when two dependencies are composed, the traces associated with them are set appropriately. For the case of composition the traces of the arguments are unified (set union) and associated with the result of the composition. In the case of loop-correction, when a dependency is created due to the LC operator, the new dependency has the same trace as the one it originated from.

For example, consider Example 6.7, the dependencies calculated for \([A_{copy,2,3}]_{dep}\) are each individually associated with \(\{A_{copy,2,3}\}\), and later on they are composed with the dependencies of \([A_{sum,3,1,2}]_{dep}\) (which are associated with \(\{A_{sum,3,1,2}\}\)). Each dependency which is part of that composition, for example, \(3 \xrightarrow{\delta_i} 3\), will be associated with the union of the two traces, i.e., with \(\{A_{copy,2,3}; A_{sum,3,1,2}\}\).

Another case from the same example is the output dependency of \(2 \xrightarrow{\delta_f} 2\), which is associated with the trace of \(\{A_{copy,1,3}\}\), and the dependency of \(5 \xrightarrow{\delta_f} 2\), which is associated with the trace of \(\{A_{copy,2,3}; A_{sum,3,1,2}\}\). These are two dependencies of different kinds, which are of the same variable, and are the results of two different traces. The former dependency results from skipping the recurrence pattern altogether, while the latter is the result of entering it.
7.4 P-TIME Complexity

The conversion algorithm described above is in P-TIME. This follows from the fact that dependency sets form a complete partial order (CPO) of polynomial height, that is, any ascending chain within it is polynomially bounded in the number of variables. In addition, the basic operations on dependency sets (composition, join) are polynomial. Therefore the algorithm’s LFP function runs in polynomial time in the number of variables of the input program.

Every node in the input flowchart can be incident in every loop, and therefore be considered a boundary node, hence the number of virtual nodes created in the course of the algorithm is bounded by $2 \cdot |V| \cdot |L|$, where $|V|$ is the number of nodes in the input graph and $|L|$ is the number of loops in the loop tree.

The running time of the algorithm is determined by a few types of operations, their complexity, and the number of times they occur.

1. Fixpoint calculation. As mentioned above, the complexity of the fixpoint calculation is polynomial in the number of variables. LFP is calculated at most once per node, including virtual nodes, for a total running time of $O(Poly(|Vars|) \cdot |V| \cdot |L|)$.

2. Dependency set composition. The size of a dependency set is $O(|Vars|^2)$, a composition considers all possible pairs of dependencies of its arguments, therefore the complexity of the composition is $O(|Vars|^4)$. A composition is done at most once for every set of three nodes, for a total running time of $O(|Vars|^4 \cdot (|V| \cdot |L|)^3)$.

Note that every arc can appear in at most one loop, therefore $|L|$ is bounded by $|E|$ keeping the above expression polynomial regardless of the number of loops.

8 Handling Resets

The action of setting a variable’s value to zero is called a reset. Considering the effects of reset actions is a substantial extension of the analysis. Consider Figure 6, clearly every execution path, also referred to as a trace, contains arcs either in ‘loop a’ or ‘loop b’ but not both, because of the preceding nondeterministic branch. Without considering reset actions, reset commands
are counted as \texttt{skip} commands, and the analysis will include traces that contain arcs from both loops, which is an overapproximation we’d like to avoid.

The terms used in this section are formally defined later on in this article, under 8.5.1.

Figure 6: Nondeterministic flowchart with two loops, only one of which gets entered depending on branch previously taken.

8.1 The Exploded Graph

One way to handle resets is to consider which variables can be zero at the entry to each node, we call this information a \textit{context}, and a node may have several contexts. Using these contexts we can create an equivalent graph that has this information embedded in its structure, and does not contain resets. We call this process \textit{exploding}, and the result, the \textit{exploded} graph.

In the example shown above, node $A$ has two contexts, $\{\text{Zero, Any}\}$ in which $a = 0$ and $b$ is not known to be zero, and $\{\text{Any, Zero}\}$ where $b = 0$ and $a$ is not known to be zero. Figure [7] shows the exploded graph derived from embedding this information in the original. A node in the exploded graph has a label which is an annotated version of the label of its progenitor node in the original graph, which indicates the specific context for this exploded node.
8.2 Abstract Interpretation – Calculating Contexts

Abstract interpretation gives a precise result when calculating contexts. Our abstract domain is composed of variable values of Zero or Any, and of states which are sets of variable assignments. Formally a context is a function $\text{Var} \mapsto \{\text{Zero, Any}\}$. An abstract state is a set of contexts, as described above.

Initially all nodes have an empty state ($\bot$), except the initial node(s) of the flowchart, which have a state containing a single context where all variables have the Any value.

The abstract interpretation algorithm iteratively makes passes over all arcs of the flowchart, and for every arc, calculates a poststate (a set of contexts) by applying the abstract transfer function of the arc’s command to
the contexts of its source node – the prestate.

At the end of each pass, states calculated for each arc are joined (set union) with the states that exist at the arc’s target node. We call this process propagation. Once a pass has been made where propagation did not change any state, the algorithm is finished, and we can pass on the context information to the explode algorithm.

This abstract interpretation is a simple case of trace partitioning as described in [9]. Our contexts are actually digested traces with elements indicating whether a variable’s value can be traced back to an input variable, a reset action, or an unbounding action (\(X:=**\)). An abstract state holds all such trace digests that are possible at its location.

It is important to note that although we have three possible origins for a variable’s value – a reset action, a program input and an unbounding action, the latter two are equivalent in our analysis (with regards to the explosion transformation), and are treated in the same way. Because of this our abstract domain contains only two values – Zero which indicates that the variable can be traced to a reset action, and Any which indicates that it can be traced either to the program’s start, or to an unbounding action.

### 8.3 Exploding the Graph – In Detail

Once contexts are known, exploding the graph is done by generating a virtual node \(v^q\) for every node-context pair \((v,q)\), and then connecting these virtual nodes with commands that are modified versions of their original commands. We denote this command modification function by \(Mod(Cmd, Ctx)\), where \(Cmd\) is the input command, and \(Ctx\) is the context of the modification.

The modified commands are generated by embedding the information from the context into the command, for example

\[
Mod(X_1 := X_2 + X_3, \{\text{Any, Zero, Any}\}) = X_1 := X_3
\]

In other words when variable \(X_2\) is known to be zero, and the other variables are not, the command \(X_1 := X_2 + X_3\) is identical to \(X_1 := X_3\). The same command with the context of \(\{\text{Any, Zero, Zero}\}\), is identical to Skip, and the target node will in this case have the context of \(\{\text{Zero, Zero, Zero}\}\). Denote the post-context of a command-context pair by \(PostCtx(Cmd, Ctx)\).

Since commands are deterministic, there is only one such context for every context and command.
Definition 8.1 (*Mod* and *PostCtx*). Figure 8 provides the definition of the *Mod* and *PostCtx* functions by listing inputs and outputs for special cases. In all unlisted cases, the functions do nothing, i.e., Mod(Cmd, Ctx) = Cmd and PostCtx(Cmd, Ctx) = Ctx. When a * appears as a context element in the *Mod* column, it means the value doesn’t matter, and when it appears in the *PostCtx* column, it means the value is preserved.

<table>
<thead>
<tr>
<th>Cmd</th>
<th>Ctx</th>
<th>Mod(Cmd, Ctx)</th>
<th>PostCtx(Cmd, Ctx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_1 := 0)</td>
<td>{\ast}</td>
<td>skip</td>
<td>{Zero}</td>
</tr>
<tr>
<td>(X_1 := X_2)</td>
<td>{\ast, Zero}</td>
<td>skip</td>
<td>{Zero, Zero}</td>
</tr>
<tr>
<td>(X_1 := X_2)</td>
<td>{\ast, Any}</td>
<td>(X_1 := X_2)</td>
<td>{Any, Any}</td>
</tr>
<tr>
<td>(X_1 := X_2 + X_3)</td>
<td>{\ast, Zero, Any}</td>
<td>(X_1 := X_2)</td>
<td>{Any, Any}</td>
</tr>
<tr>
<td>(X_1 := X_2 + X_3)</td>
<td>{\ast, Any, Zero}</td>
<td>(X_1 := X_2)</td>
<td>{Any, Any}</td>
</tr>
<tr>
<td>(X_1 := X_2 + X_3)</td>
<td>{\ast, Zero, Zero}</td>
<td>skip</td>
<td>{Zero, Zero, Zero}</td>
</tr>
<tr>
<td>(X_1 := X_2 + X_3)</td>
<td>{\ast, Any, Any}</td>
<td>skip</td>
<td>{Any, Any, Any}</td>
</tr>
<tr>
<td>(X_1 := X_2 + K)</td>
<td>{\ast, \ast}</td>
<td>skip</td>
<td>{Any, \ast}</td>
</tr>
<tr>
<td>(X_1 := X_2 \ast X_3)</td>
<td>{\ast, Zero, \ast}</td>
<td>skip</td>
<td>{Zero, Zero, \ast}</td>
</tr>
<tr>
<td>(X_1 := X_2 \ast X_3)</td>
<td>{\ast, \ast, Zero}</td>
<td>skip</td>
<td>{Zero, \ast, Zero}</td>
</tr>
<tr>
<td>(X_1 := X_2 \ast X_3)</td>
<td>{\ast, Any, Any}</td>
<td>skip</td>
<td>{Any, Any, Any}</td>
</tr>
<tr>
<td>(X_1 := X_2 \ast K)</td>
<td>{\ast, Zero}</td>
<td>skip</td>
<td>{Zero, Zero}</td>
</tr>
<tr>
<td>(X_1 := X_2 \ast K)</td>
<td>{\ast, Any}</td>
<td>skip</td>
<td>{Any, Any}</td>
</tr>
<tr>
<td>(X_1 := **)</td>
<td>{\ast}</td>
<td>skip</td>
<td>{Any}</td>
</tr>
</tbody>
</table>

Figure 8: Definition of *Mod* and *PostCtx*

The process of modifying commands in this way results in a graph which has more nodes and more arcs, but has no reset actions, and represents a program which is equivalent to the original one in the following sense: both programs have the same set of transition sequences (Definition 2.6), as long as for the exploded graph states (Definition 2.3) we list the node name without its context qualification as the program location.

An arc connects a virtual node \(u^p\) to the virtual node \(v^q\) when \((u, v)\) is an arc in the source graph, and \(v = PostCtx(\text{Inst}(u, v), p)\).

Once the algorithm has finished running, context information can be discarded, and analysis can commence on the exploded graph. As seen in the example in Figure 7, the exploded graph may contain multiple final nodes, if the final node was determined to have several contexts, and in this case
the analysis will provide results for traces ending at any of these final virtual
nodes.

**EXAMPLE 8.2** (transition sequence equivalence). Consider the flowchart
in Figure 1 and the following trace: \(10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 41 \rightarrow 411 \rightarrow 40 \rightarrow 42 \rightarrow 40 \rightarrow 20 \rightarrow 30 \rightarrow 50\).

Assuming the initial state of \(\{a = 1; b = 2; c = 3; d = 4; e = 5; g = 7; h = 8; i = 9; j = 10; u := 20\}\), this trace would create the following transition
sequence (the commands are specified above the arrows just for reference, they are not part of the transition sequence):

\[
(10, \{a = 1; b = 2; c = 3; d = 4; e = 5; g = 7; h = 8; i = 9; j = 10; u = 20\}) \\
\overset{a := b}{\longrightarrow}(20, \{a = 2; b = 2; c = 3; d = 4; e = 5; g = 7; h = 8; i = 9; j = 10; u := 20\}) \\
\overset{c := c + d}{\longrightarrow}(30, \{a = 2; b = 2; c = 7; d = 4; e = 5; g = 7; h = 8; i = 9; j = 10; u := 20\}) \\
\overset{d := d + c}{\longrightarrow}(40, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 7; h = 8; i = 9; j = 10; u := 20\}) \\
\overset{g := g + h}{\longrightarrow}(41, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 9; j = 10; u := 20\}) \\
\overset{\text{reset} i}{\longrightarrow}(40, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 0; j = 10; u := 20\}) \\
\overset{g := g + h}{\longrightarrow}(41, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 9; j = 10; u := 20\}) \\
\overset{\text{reset} i}{\longrightarrow}(40, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 0; j = 10; u := 20\}) \\
\overset{i := g}{\longrightarrow}(42, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 0; j = 10; u := 20\}) \\
\overset{j := i}{\longrightarrow}(40, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 8; j = 8; u := 20\}) \\
\overset{\text{reset} e}{\longrightarrow}(20, \{a = 2; b = 2; c = 7; d = 11; e = 0; g = 0; h = 8; i = 8; j = 8; u := 20\}) \\
\overset{e := e + d}{\longrightarrow}(30, \{a = 2; b = 2; c = 18; d = 11; e = 0; g = 0; h = 8; i = 8; j = 8; u := 20\}) \\
\overset{u := g}{\longrightarrow}(50, \{a = 2; b = 2; c = 18; d = 11; e = 0; g = 0; h = 8; i = 8; j = 8; u = 0\})
\]

Which is also supported by the exploded graph (Figure 2) (we indicate
the context beside the program location just for reference, it is not part of
the transition sequence):
(10, \{a = 1; b = 2; c = 3; d = 4; e = 5; g = 7; h = 8; i = 9; j = 10; u = 20\})

\[
\begin{align*}
\text{reset}_a & \rightarrow (20, \{a = 2; b = 2; c = 3; d = 4; e = 5; g = 7; h = 8; i = 9; j = 10; u := 20\}) \\
\text{reset}_c & \rightarrow (30, \{a = 2; b = 2; c = 7; d = 4; e = 5; g = 7; h = 8; i = 9; j = 10; u := 20\}) \\
\text{reset}_d & \rightarrow (40, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 7; h = 8; i = 9; j = 10; u := 20\}) \\
\text{reset}_g & \rightarrow (41, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 15; h = 8; i = 9; j = 10; u := 20\}) \\
\text{reset}_i & \rightarrow (41\{g\}, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 9; j = 10; u := 20\}) \\
\text{reset}_i & \rightarrow (40\{g, i\}, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 0; j = 10; u := 20\}) \\
\text{reset}_g & \rightarrow (41\{i\}, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 8; h = 8; i = 0; j = 10; u := 20\}) \\
\text{reset}_j & \rightarrow (42, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 8; h = 8; i = 8; j = 10; u := 20\}) \\
\text{reset}_j & \rightarrow (40, \{a = 2; b = 2; c = 7; d = 11; e = 5; g = 0; h = 8; i = 8; j = 8; u := 20\}) \\
\text{reset}_e & \rightarrow (20\{e\}, \{a = 2; b = 2; c = 7; d = 11; e = 0; g = 0; h = 8; i = 8; j = 8; u := 20\}) \\
c & \rightarrow (30\{e\}, \{a = 2; b = 2; c = 18; d = 11; e = 0; g = 0; h = 8; i = 8; j = 8; u := 20\}) \\
\text{skip} & \rightarrow (50\{e\}, \{a = 2; b = 2; c = 18; d = 11; e = 0; g = 0; h = 8; i = 8; j = 8; u = 0\})
\end{align*}
\]

Note that the transition sequences are identical despite the fact that the command on the transition 40 \rightarrow 41\{i\} and the command on the transition 30 \rightarrow 50 are different between the two traces.

### 8.4 Complexity – Exponential

It is easy to provide a contrived example which results in exponential running time of the explosion algorithm. Simply take a flowchart program with \(n\) variables, which nondeterministically sets each variable either to zero, or to an unbounded value. The result of the explosion algorithm on such a program would create \(2^n\) copies of the final node of the flowchart.

Also, since the algorithm solves a decision problem known to be PSPACE-hard (\([4]\)) we do not expect a polynomial-time solution anyway.

### 8.5 Abstract Interpretation – Formal Definitions

#### 8.5.1 Abstract Domain

1. **Variables** – The set of variables \(Var = \{x_1, ..., x_n\}\).
2. **Abstract Values** – The set of abstract values $\mathbb{V} = \{\text{Any}, \text{Zero}\}$.

3. **Contexts** – A context is a function which is an assignment of abstract values to all program variables. The set of contexts $\mathbb{C} = \text{Var} \mapsto \mathbb{V}$.

4. **States** – A state: $s \in \mathbb{C}$ is a set of contexts. The set of all states is $\mathbb{S} = \mathcal{P}(\mathbb{C})$.

### 8.5.2 Transfer Function

The basic step in the process of abstract interpretation is applying a transfer function specific to a concrete command, on an abstract state (the prestate), and obtaining a new state (the poststate). We define a transfer function which is applied point-wise to all contexts contained in the prestate. The set of the resulting contexts is the poststate.

A transfer function is of the form

$$f : \text{Command} \mapsto \text{Context} \mapsto \text{Context}$$

i.e., it is a function that takes a command and a context as input, and produces a context as output. In the following list we use the notation of $\llbracket \text{Cmd} \rrbracket(C)$ to denote the application of the transfer function on the command $\text{Cmd}$ and the context $C$.

Also, we denote by $C(i)$ the abstract value associated with variable $i$ in context $C$, and we denote by $C[r \mapsto v]$ the updated context, i.e., a context which is identical to $C$ except that it maps the variable $r$ to the abstract value $v$.

The operators ($+$) and ($*$) between abstract values are defined as follows:

- $a + b = \text{Any}$ if either $a = \text{Any}$ or $b = \text{Any}$, otherwise $\text{Zero}$
- $a * b = \text{Any}$ if both $a = \text{Any}$ and $b = \text{Any}$, otherwise $\text{Zero}$

1. **Skip**

   $\llbracket \text{Skip} \rrbracket(C) = C$

2. **Identity Assignment**

   $\llbracket X_r := X_l \rrbracket(C) = C[r \mapsto C(l)]$
3. **Addition Assignment**

\[ X_r := X_{l_1} + X_{l_2} \] 
\( (C) = C[r \mapsto C(l_1) + C(l_2)] \)

4. **Multiplication Assignment**

\[ X_r := X_{l_1} \ast X_{l_2} \] 
\( (C) = C[r \mapsto C(l_1) \ast C(l_2)] \)

5. **Reset Command**

\[ X_r := 0 \] 
\( (C) = C[r \mapsto \text{Zero}] \)

6. **Unbound Command**

\[ X_r := ** \] 
\( (C) = C[r \mapsto \text{Any}] \)

### 9 The Application Software

The algorithms described in this article for analysing flowchart programs (7.2, 8.5) have been implemented in a software application tool with a graphical user interface. The tool is an integrated environment, which allows the user to both enter and edit source code for a flowchart program, and to compile and view a graphical representation of that flowchart, along with the results of the analysis. The tool allows the user to view both the original, and the exploded versions of the flowchart (see Section 8.1). The tool can also accept core language programs as input, and converts them to flowcharts for display and analysis.

All algorithms are implemented in **F#** (*F-Sharp*), which is an advanced object-oriented functional language, based on **OCaml**, while the GUI is implemented in **C#** over the WPF framework (*Windows Presentation Foundation*). All the algorithmic code is implemented in a pure functional way.

After compiling and analyzing a program, the dependency results are shown on the right side of the window. Clicking on a dependency entry will highlight the arcs on the graphical representation of the graph, which are in the trace of that dependency.

Code Snippet 4 shows an example of source code for a flowchart program. The source is structured as a sequence of alternating locations and commands. The `end` keyword signifies the end of a sequence, and the optional start of another. Every instruction becomes an arc in the flowchart, with the instruction as its label, between nodes corresponding to the locations before and after the instruction. The `end` pseudo-instruction is useful...
to describe code that is not linear. Loops are defined by enclosing sequences in \texttt{loop } \texttt{X_i ... end loop} commands, where \texttt{X_i} is the loop bound variable. \texttt{OCaml}-style source code comments are supported ((* ... *)).

The formal definition of the input formats (flowchart and core) are included in the help file of the tool.

**Code Snippet 4** Sample flowchart program source code

\begin{verbatim}
10 c:=a (* 10->20 *)
20 end
loop e
  20 c:=a+b
  30 b:=c
  20 d:=c
  50 end (* 20->30->20->50 *)
end loop
50 skip (* 50->60 *)
60 end (* final node *)
\end{verbatim}

**References**


48


$$x_1 := x_3;$$

loop $x_5$ {
    $$x_2 := x_3;$$
    $$x_3 := x_1 + x_2$$
};

$$x_4 := x_3$$

Figure 4: This example illustrates how a structured program might be translated into a flowchart. The loop structure consists of the root $R = \{(P_1, P_2), (P_4, P_5)\}$, and the loop $L = \{(P_2, P_3), (P_3, P_4), (P_4, P_2)\}$ with bounding variable $x_5$. 
\( X_1 := 0; \)
\( X_2 := **; \)

\[
\text{loop } X_1 \{ // \text{ Loop } A_2 \\
\quad X_3 := X_1 + 1; \\
\quad X_1 := X_3; \\
\quad \text{loop } X_3 \{ // \text{ Loop } A_3 \\
\quad \quad X_4 := X_3 \times 2 \\
\quad \}; \\
\quad \text{loop } X_3 \{ // \text{ Loop } A_4 \\
\quad \quad X_4 := X_3 \times 2 \\
\quad \}; \\
\quad X_2 := X_3 \\
\}; \\
\text{loop } X_2 \{ // \text{ Loop } A_1 \\
\quad X_3 := X_1 + 1 \\
\} \]

Figure 5: The following program has nested loops, forming the tree depicted below the program text, where \( A_0 \) represents the non-looping part of the program.