1 Introduction

It’s often useful to ask questions about programs without actually running them. For example, we might want to know if a program contains some class of bugs such as division by zero or memory access errors. For compilers static information is vital for good quality machine code generation. Another class of questions is about cost of running a program. For instance, we might want to know if the program will terminate, or how much memory it uses, or we might need to approximate its run-time. Approximation of run-time is a useful in distributed systems for optimally planning execution of tasks. It’s well known that in general answering questions about non-trivial properties of programs is undecidable. However, this doesn’t mean that we cannot answer questions about wide class of programs or that we cannot provide approximations of answers we want to get. We call analysis of programs that is performed without actually executing the program a static program analysis. We will focus on static analysis of imperative programs.

High-level programming languages are often quite complicated and present significant challenge if we want to analyse the code written in one directly. The more syntactic cases to handle the more difficult the analysis problem becomes. This isn’t to say that static analysis can’t be done directly on high-level language, it often is, and many static program analysis tools, that focus on detecting programmer errors, are designed in such way. However, in order to reduce complexity, a high-level program code is often converted to a simpler, easy to handle, but semantically equivalent, form. This simpler form is either called intermediate code or representation.

In this work we do not work on a concrete high-level language, but assume that program is represented as a sequence of simple statements. This could stand as an
intermediate language for some high-level programming language. We will not specify this language formally and will only present informal description. Statements of the language are denoted with $s$, and variables with $x$ and $y$. The language has no structured control flow, and only control flow is either implicit from statement to the immediately following statement or is performed via (conditional) jumps to statement labels. For a statement $s_i$ we sometimes jump to the index $i$ to denote jump to that statement. For some statement $(x_1, \ldots, x_n) = \text{op}(y_1, \ldots, y_m)$ we say that the statement defines variables $x_1, \ldots, x_n$ and uses variables $y_1, \ldots, y_m$. All variables defined by a statement must be pair-wise different. Program is defined as a sequence of statements and is well-formed in the following sense: there are no jumps to undefined labels, and all variables are defined before they are use.

Rest of this work is structured as follows. In Sec. 2 we present overview of static program analysis techniques. We start with an overview of classic iterative data-flow analysis framework. In the second part of the section we explore constraint-based approach to static analysis and see how it relates to data-flow analysis. In Sec. 3 we give a brief overview of static cost analysis, and in Sec. 4 we conclude.

## 2 Static program analysis

This section covers the basics of static program analysis. We provide the notation and introduce the concept of static data-flow analysis for gathering static information about the flow of data in programs. We present live variables analysis, as an example of data-flow analysis, that, for every program point, computes the set of variables that the program might read in the future. In the second part of this section we introduce constraint-based program analysis. As example we present live variables analysis as a problem that can be solved with constraint-based approach. Overview of the data-flow analysis is based on [1], and the constraint-based analysis on [2].

To keep things simple we are imposing some restrictions on the programs we are analysing. We assume that every program is composed of a sequence of statements $s_0, s_1, \ldots, s_n$, where $n \geq 0$, such that the execution of the program starts with statement $s_0$ and always terminates executing statement $s_n$. The latter restriction is not limiting as every program that contains multiple exit statements can be converted to such form by introducing a new exit statement and replacing the old exit statements with jumps to the new statement. The second simplification we make is that the program does not contain functions. While it’s possible to eliminate all functions from a program, via code duplication and variable substitution, it’s often not feasible as the target program can grow very large. One approach in program analysis for handing functions is to approximate the behaviour of functions very roughly in a context independent manner, i.e., the abstract behaviour of a function is the same on every call.

Often the control flows from a statement $s_j$ to the immediate next statement $s_{j+1}$,
but most useful programs have more complicated control flow structure – individual statements can conditionally jump to different locations. If control can jump from a statement $s$ to a statement $s'$ we call $s$ the predecessor of $s'$ and $s'$ the successor of $s$. For every statement $s$ the set of successors is denoted with $\text{succ}[s]$ and the set of predecessors with $\text{pred}[s]$. A directed graph is called a control-flow graph of a program $s_0, \ldots, s_n$ if the nodes are the statements of the program and the edges $s \to s'$ of the graph denote that statement $s'$ is a successor of statement $s$.

### 2.1 Data-flow analysis

One of the most common static program analysis is so called data-flow analysis. Data-flow analysis refers to a class of techniques that answer questions about flow of data along programs execution paths. Many of the program analysis that are required for code optimizations are covered by data-flow analysis. Only a handful of examples include: live variables, reaching definitions, constant propagation, available expressions, and anticipated expressions. We shall only look closely at live variables analysis, but all of the named examples are explored in detail in [1].

The goal of data-flow analysis is to associate a data-flow value with every single program point. Data-flow value represents an abstraction of the set of all possible program states that can be observed for that point. For a particular analysis a set of all possible data-flow values $V$ is called the domain of that analysis. For instance, the domain of live variables analysis is the power set of all variables that occur in the program and, for every program point, the data-flow value denotes the variables that might be read from the given point on.

For every program statement $s$ we denote the data-flow value immediately before and after executing the statement with $\text{in}[s]$ and $\text{out}[s]$ respectively. The goal of data-flow analysis is to find a solutions to the set of constraints imposed on the input and output data-flow values. Every statement of the program imposes some constraints on the input and output values of the statement itself but, additionally, the control flow-graph of the program imposes some more constraints. The particulars of the constraints depend of the analysis.

#### 2.1.1 Lattice-based framework

Thus far we haven’t said much about the particulars of the data-flow values or discussed if data-flow problem can be solved for any kinds of values and constraints. Turns out that there exists a class of values that allows for the data-flow problem to be solved efficiently. The abstraction that describes this class is called a semilattice.

**Definition 1.** A semilattice is a pair $(V, \wedge)$ that consists of a set $V$ and a binary meet operator $\wedge$ such that for every $x, y, z \in V$ the meet operator is: 1) idempotent $x \wedge x = x$; 2) commutative $x \wedge y = y \wedge x$; and 3) associative $x \wedge (y \wedge z) = (x \wedge y) \wedge z$. 

3
Every semilattice has a top element, denoted $\top$, such that for all $x \in V$ we have $x \land \top = \top$. Semilattice induces partial ordering $x \leq y$ iff $x \land y = x$. It’s relatively straightforward to show that the induced $\leq$ operator is in fact a partial order.

**Definition 2.** A data-flow analysis framework is a quadruple $(D, V, \land, F)$ that consists of 1) a direction of the data flow $D$, which is either forward or backward; 2) a semilattice $(V, \land)$ with domain $V$ and meet operator $\land$; and 3) a family $F \subseteq \{f | f : V \rightarrow V\}$ of transfer functions.

Two restrictions are imposed on the framework: the family of transfer functions is closed under function composition, and the framework is monotone – for every $x, y \in V$ and $f \in F$ if $x \leq y$ then $f(x) \leq f(y)$. Those restrictions are sufficient for there to exist an iterative algorithm for solving data-flow problems in this framework.

To solve a given data-flow problem for an arbitrary program we need that, for every statement $s$ of the program, there exists a transfer function $f_s \in F$. In case of forward (backward) data-flow problem we require that a constant value $v_{\text{ENTRY}} \in V$ ($v_{\text{EXIT}} \in V$) is fixed for specifying the data-flow value of the program before (after) executing it.

The algorithm for solving data-flow problem for a program, consisting of statements $s_0, \ldots, s_n$, finds values $\text{IN}[s]$ and $\text{OUT}[s]$ for every statement $s$ of the program. Two algorithms are provided: one for solving forward data-flow problems (Fig. 1a), and other for solving backward data-flow problems (Fig. 1b). It can be shown that if the algorithm terminates for a well-formed data-flow analysis framework then the input and output sets of all statements satisfy the constraints. If the semilattice $(V, \land)$ is of finite height then the algorithm always terminates for any program. Lattice is said to have a finite height if every strictly descending chain $v_1 < v_2 < \ldots < v_n$ is finite.

It’s important to note that multiple solutions might, and often do, exist for a given set of data-flow equations. In general case the iterative algorithms provide only a conservative estimation of the precise solution. However, if the analysis framework is distributive then the analysis result is precise. Framework $(D, V, \land, F)$ is called distributive if $f(x \land y) = f(x) \land f(y)$ for any $f \in F$ and $x, y \in V$. Live variables analysis is distributive, but constant propagation is a well known non-distributive analysis.

---

**Figure 1** Iterative data-flow algorithms

\[
\begin{align*}
\text{IN}[s_0] &= v_{\text{ENTRY}} \\
\text{OUT}[s_0] &= \top \\
\text{OUT}[s_i] &= \top & (0 < i \leq n) \\
\text{while } \text{changes to any OUT occur} & \text{ do} \\
& \text{for } 0 < i \leq n & \text{do} \\
& \text{IN}[s_i] &= \land_{s \in \text{pred}[s]} \text{OUT}[s] \\
& \text{OUT}[s_i] &= f_{s_i}(\text{IN}[s_i]) \\
\text{end for} \\
\text{end while} \\
\text{while } \text{changes to any IN occur} & \text{ do} \\
& \text{for } 0 \leq i < n & \text{do} \\
& \text{OUT}[s_i] &= \land_{s \in \text{succ}[s]} \text{IN}[s] \\
& \text{IN}[s_i] &= f_{s_i}(\text{OUT}[s_i]) \\
\text{end for} \\
\text{end while}
\end{align*}
\]

(a) Forward data-flow algorithm

(b) Backward data-flow algorithm
2.1.2 Example: live variables analysis

As previously mentioned, live variables analysis is a data-flow analysis with aim to find out, for every program point, a set of live variables. Variable is said to be alive at a given program point if it might be read, with no intervening assignments, after that point during the execution. Live variables information is, in general case, not computable, but we can overestimate and say that some variables are alive while in practice they would never be read. Accurate live variables information is essential for good register assignment. Namely, if a variable is alive at assignment of some other variable then the values of those two variables can not be stored in the same register. Conversely, if two variables are not alive at the same time at any program point, then the values of those variables can be stored in the same register.

Live variables analysis is perfect fit for lattice-based data-flow framework. The data-flow values for live variables analysis are sets of variables. Hence for the data-flow domain we can use the power set of all variables. The data-flow information travels backwards in the control-flow graph: if current statement reads a variable then at every preceding point that variable needs to be alive, and if a statement defines a variable that variable is dead at all preceding points. Following the same reasoning the meet operator of the lattice is union of sets and the top element \( \top \) is the set of all variables.

At the exit of the program no variable is alive.

To finish constructing the data-flow analysis framework we need to define the class of transfer functions. In general, for every statement \( s \) in the form \( (x_1, \ldots, x_n) = \text{op}(y_1, \ldots, y_m) \), we define a set of variables used by the statement \( \text{use}_s = \{y_1, \ldots, y_m\} \), and the set of variables defined by the statement \( \text{def}_s = \{x_1, \ldots, x_n\} \). The transfer function \( f_s \) for statement \( s \), defined as \( f_s(x) = \text{use}_s \cup (x - \text{def}_s) \), kills all variables defined by the statement and marks all variables used by the statement alive. Note that if a variable is defined and used by the same statement, such as \( x = x + 1 \), the variable is alive before that statement.

Given a program \( s_0, \ldots, s_n \) with variables occurring from the set \( \{x_1, \ldots, x_m\} \) the lattice-based framework supplies us with concrete algorithm (Fig. 2) for solving the live variables problem. A quick side node – the algorithm finds precise solution to the set of constraints, but it does not, in general case, give use the best possible liveness information. We have previously already noted that solving live variables problem is not computable. However, the algorithm itself can be improved. Namely, the current algorithm considers all possible paths in the control-flow graph but in the actual program some paths might
not be possible to take. This is to say that the analysis is not path-sensitive. Typically data-flow analysis are path-insensitive.

2.2 Constraint-based program analysis

The constraints-based approach to program analysis splits the problem into two: constraints generation, and constraint resolution. The constraint generation step produces a set of constraints from a program text that stands as a specification of the information we want to know about the program. The constraint resolution then computes the desired information. According to Aiken [2] the constraint-based analysis paradigm is appealing for three primary reasons: 1) Constraint separate specification and implementation; constraint generation is the specification of the analysis, and constraint resolution is the implementation. 2) Constraints yield natural specification; constraints are usually local and only the conjunction of all constraints describes global properties of the program. 3) Constraints enable sophisticated implementation; constraint problems have rich theory that can be used in implementation of a particular analysis.

Data-flow analysis framework provides a generic and often very efficient framework for solving a wide class of static analysis problems. We can view the iterative data-flow algorithms to solve a set of constraints that’s generated implicitly by the algorithm itself. For example, for a program \( s_0, \ldots, s_n \), the live variables algorithm implicitly generates the following equality constraints:

\[
\begin{align*}
\text{out}[s_i] &= \bigcup_{s \in \text{succ}[s_i]} \text{in}[s] \\
\text{in}[s_i] &= \text{use}_{s_i} \cup (\text{out}[s_i] - \text{def}_{s_i})
\end{align*}
\]

where \( 0 \leq i < n \) and variables are denoted with \( \text{in}[s_i] \) and \( \text{out}[s_i] \). Solving the system consists of finding assignments to the variables such that all of the equalities hold. This system of set constraints can have multiple solutions, but in general we are looking for the best one.

Thus far we haven’t said anything about the (im)possibility of solving particular constraints. Naturally not every class of constraints is solvable, and many of the classes are not solvable in efficient time. For example, satisfiability of unrestricted set constraints (union, intersection, complement, and \( n \)-ary constructors) is known to be \( \text{NEXPTIME-complete} \) problem, but many restricted forms of set constraints can be solved in polynomial time.

One of the best-known methods, that’s directly covered by the constraint-based analysis framework, is finite lattice method introduced by Cousot and Cousot in the seminal paper on abstract interpretation [7]. In this paradigm a finite lattice \( V \) is designed, and the program analysis is expressed as a system of recursive equations in the form \( x_1 = \sigma_1(X), \ldots, x_n = \sigma_n(X) \) where \( X = \{x_1, \ldots, x_n\} \) is a set of variables and every
\( \sigma_I : V^{|X|} \rightarrow V \) is a monotonic function\(^1\). It’s well known that such set of equations can be solved with generic iterative fixed point algorithm that computes the least solution [7] to such equations. This provides us with an example of constraint-based analysis – live variable constraints can be solved by this iterative method, and the solution will be the best one. More generally; every finite lattice based data-flow problem can be solved using finite lattice method by splitting the problem into constraint generation and constraint resolution.

Constraint-based approach turns out to also be useful in type inference, i.e., deriving a type of given function. The usual approach for type inference for lambda terms consists of generating a set of constraints from the term, and resolving the constraints to get the type of the term. Interested reader can find Hindley-Milner type inference algorithm in work by Wand [13].

### 3 Cost analysis

Information about the execution cost of programs, or the amount of resources that the execution will require, is useful for various purposes. For instance, in case of Java bytecode, we might not want to execute the code that has too large cost requirement in terms of computing resource. In parallel systems knowledge about cost of tasks can be used to schedule execution of them. Cost, or resource, analysis is not only limited to approximating the run-time cost. We might also be interested in network usage, stack usage, heap usage, or even use of monetary resources if the code might send text message or connect to internet.

Cost analysis is a well studied problem in context of declarative programming languages [11, 9]. However, the cost analysis of imperative languages has been very limited and has been mostly restricted to worst-case execution time estimation of high-level languages. Recently a body of research has emerged on cost analysis of Java bytecode [3, 4]. Research project Mobile Resource Guarantees [5] established a proof-carrying code [10] framework for guaranteeing resource consumption. The Mobility, Ubiquity and Security research project [6] also considers resource consumption as one of the central interests for proof-carrying code.

This section is mostly based on work on cost analysis of Java bytecode by Albert et al. in [4]. However, we sidestep all the issues that arise in converting the bytecode to representation that is more convenient to analyse. Instead, for clarity, we directly operate on a rather abstract intermediate representation that differs greatly from the representation used in [4].

The analysis no longer operates on a program as in Sec. 2 and instead we operate on an \( n \)-ary procedure \( p \) with formal parameters \( x_1, \ldots, x_n \) and body that consists of

\(^1\)An \( n \)-ary function is said to be monotonic if for all \( x_1 \leq y_1, \ldots, x_n \leq y_n \) then \( f(x_1, \ldots, x_n) \leq f(y_1, \ldots, y_n) \).
statements \(s_0, \ldots, s_m\) with same restrictions. The behaviour of the procedure depends on values of the formal parameters. We limit the analysis to a single procedure that does not return a value and does not call other procedures.

The goal of cost analysis is to approximate the execution cost of a procedure as a function from parameter sizes. Size is always integer valued and in case of integer valued symbols denotes the value of the symbol. If the structure of input symbol is more complicated the cost might denote something different. For example, the size may refer to longest reachable path in a tree structure, or length of a list.

Cost analysis of a procedure is performed in three steps: 1) infer size relations that constrain the possible sizes of variables, 2) derive a set of symbols that are relevant to the cost of running the procedure, and 3) from the size relations and relevant symbols build cost relations. The produced cost relations can, in many cases, be simplified to the point of statically deriving an upper and lower cost bounds or a closed form solution. Such simplifications have been well-studied in the field of algorithmic complexity.

3.1 Size relations

**Definition 3** (Size relation). For every \(1 \leq i \leq n\) let \(a_i\) be an integer constant, \(x_i\) a variable, and \(R \in \{=, \leq, <\}\) a relation. We call a conjunction of linear constraints \(a_0 + a_1x_1 + \ldots + a_nx_n R 0\) a size relation and denote it with \(\varphi\).

Obtaining size relations is imperative for cost analysis. In particular, they are essential for defining a cost of a statement in terms of it’s successors. In general, we can use various measures to denote size, but in this particular case we only consider the case of integer variables. Our aim is to derive, for every program point, a size relation between the variables of the program. Inferring the size relations is not straightforward, but is a well studied problem [8].

Inferring size relations is done in two steps. The first step is to annotate each arc of a control flow graph of the procedure with local constraints. This is performed via abstract interpretation where each statement adds restrictions to outgoing arcs. For instance, statement \(s_i : x = y\) imposes equality constraint \(x = y\) for the arc \(s_i \rightarrow s_{i+1}\). Note that we use the name of the variable to refer to its size. Branching statements add constraints to all outgoing arcs. For example, conditional statement \(s_i : \text{jmp} l t j x y\), that jumps to statement \(s_j\) if \(x < y\) and the next statement otherwise, imposes constraint \(x \geq y\) to false arc and constraint \(x < y\) to true arc. We assume that every arc of the procedure is annotated with a constraint.

The constraints that we generated via abstract interpretation are local to arcs and do not reflect all linear relations between program variables. Hence, the second step is to resolve the system of constraints to produce size relations that reflect linear constraints between all variables. Unfortunately, this constraint system is not solvable using tools at hand – the domain is not finite and there seems to be no obvious way
to construct a finite height lattice. We will not dwell into the details of the solution but the idea is to implement iterative algorithm similar to forwards data-flow analysis algorithm. However, during the iteration the algorithm makes sure to apply *widening operator* [7] in every loop to guarantee termination. Details of the implementation [8] are quite complicated and get into computational geometry. For every statement $s$ let $\varphi_s$ denote the size relation between all variables after executing statement $s$. We write $\varphi_s(\mathcal{X})$, where $\mathcal{X}$ is a set of variables, to denote size relations between only variables $\mathcal{X}$ at statement $s$.

### 3.2 Cost relations

The aim of cost analysis is to find how the execution cost of a procedure depends on its input sizes. The cost function for statement $s$ takes the form $C_s : \mathbb{Z}^n \rightarrow \mathbb{N}_{\infty}$, where $\mathbb{Z}$ is the set of integers and $\mathbb{N}_{\infty}$ is the set of natural numbers extended with a special value $\infty$ to denote unbounded cost. The measure of cost is left abstract and the analysis does not depend on which concrete measure is used.

#### 3.2.1 Minimizing domain of cost function

The first step of the analysis is to minimize the number of variables that need to be taken into account by every cost function. Some of the variables, or even formal parameters, might not influence the cost of running the procedure. For example, an accumulating parameter that control flow does not depend on does not influence the cost of running the procedure.

We make an assumption that executing each individual statement adds a constant cost to executing the procedure. This is reasonable assumption to make as the intermediate representation of code is usually assumed to be relatively simple. For example, the cost of adding two numbers does not depend on the values of those numbers. Thus, the only variables that affect the cost of executing a procedure are those that may influence the control flow. If our language had other procedures then parameters to calls should be considered too.

Computing safe approximation of the set of variables affecting the control flow of the program is a well studied problem in *program slicing* [12]. The problem at hand is somewhat simpler as we don’t need to modify the program to delete redundant statements, but only collect sets of variables. We denote the set of *relevant variables* before executing a statement $s$ with $\mathcal{S}$ and after executing $s$ with $\mathcal{S}$. Sets of relevant variables can be found, for example, using finite lattice method.
3.2.2 The cost relation

The cost function $C_s : \mathbb{Z}^n \rightarrow \mathbb{N}_\infty$ for statement $s$ is defined in terms of cost relation which consists of a set of cost equations. The intuitive idea is that for every statement $s$ we generate: 1) a cost equation that denotes sum of executing that statement $s$ and its continuation; and 2) a continuation as a cost of executing one of the successor statements. The cost of statement continuation is defined conditionally depending on which branch is taken. For defining the continuation cost let, for any edge $s \rightarrow s'$ of the control flow graph, $\text{guard}(s, s')$ denote the condition under which the given edge is taken during the evaluation. The guards are assumed to be mutually exclusive and at least one of the guards needs to be taken. Let $T_s$ be constant denoting the cost of executing only statement $s$.

Definition 4 (Cost relation). Let $s$ denote a statement of a procedure, and let $\{s_1, \ldots, s_k\}$ denote the set of successors of $s$. We generate cost relation for statement $s$ as follows:

$$
C_s(\emptyset) = T_s + \sum_{i=1}^{k} \phi(s_1 \cup \ldots \cup s_k) \quad \text{guard}(s, s_i)
$$

The size relationship between relevant variables is attached to the cost equation. Cost functions only take relevant variables as parameters. The continuation cost function takes relevant variables of all successors as argument. Note that it might be that $s_1 \cup \ldots \cup s_k \not\subseteq \emptyset$ as statement $s$ can define a new variable that’s relevant to some successor. The cost of running a procedure that consists of statements $s_0, \ldots, s_n$ is hence defined with function $C_{s_0}$.

The cost $T_s$ of statement $s$ depends on the chosen cost model. If we are only interested in finding out the number of statements the code will execute, then $T_s$ can be defined to be equal to 1 for all statements. For more accurate cost model profiling can be used to associate some estimation of run-time with every statement.

3.3 Solving cost relations

We will summarise the methodology proposed in [4] for solving and approximating cost relations. The cost relations generated for a procedure in the previous section allow us to reason about the cost of running the procedure, but the described relations are recursive and each relation only models local cost. We wish to find closed form solution, or approximation, of the cost function which measures the cost of the entire procedure. This can be done in two steps: 1) we eliminate existential variables, i.e., variables that do not occur in the left hand side of the equation; and 2) the obtained
relations are solved or approximated using existing tools. The first step produces so-called *recurrence equations* and solving them is a well researched problem.

Given existential variable $y$, a size relation $\varphi$, and a set of variables $\bar{x}$, we denote by $\text{solve}(y, \varphi, \bar{x})$ the function which returns an expression $e$, such that $FV(e) \subseteq \bar{x}$ and $\varphi \vDash (y = e)$. All free variables of the expression need to occur in the specified set, and for every assignment of variables, that satisfies $\varphi$, the expression must be equal to variable $y$. The result of the function can be $y$ itself if no satisfying expression is found. This function allows us to eliminate some, if not all, existential variables by replacing them with resulting expression. We can use this method to eliminate an existential $y$ only if it can be expressed as linear combination of variables from $\bar{x}$.

In more complicated cases the size analysis might not be able to derive precise value of the existential but might be able to provide us with interval in which the values of the variable must range. The function $\text{interval}(\varphi, y, \bar{x})$ returns either: an interval $[e_1, e_2]$, such that $FV(e_1) \cup FV(e_2) \subseteq \bar{x}$ and $\varphi \vDash (e_1 \leq y \leq e_2)$; or the same variable $y$. Using intervals we can approximate either upper or lower bound of the cost function. No explanation was supplied in [4] what is done with existential variables that cannot be eliminated with the given method. It’s either implicitly assumed that interval method removes all existentials or the given set of constraints in not solvable.

Derived recurrence relations can be solved or approximating using existing tools (Mathematica, Maxima, Maple, Matlab). In general case no closed form solution can be derived for recurrence equations. However, it’s always possible estimate the cost function with either upper or lower bound. To compute, say, lower bound we replace existential variables with lower bound, and compute minimum of all the choices.

In closing we would like to point out that in follow-up work to [4] by Albert et al. [3] it’s argued that recurrence relations are not appropriate for estimating cost relations for following reasons: 1) cost relations are non-deterministic, 2) size analysis provides inexact constraints, and 3) in general cost functions have multiple arguments. In [4] cost relations are approximated directly without converting them to recurrence relations. The work presented there is more focused on solving the particular constraints than program analysis and, unfortunately, is beyond the scope of this work.

## 4 Conclusions

In this work we provided overview of static program analysis, and explored cost analysis of imperative programs. We provided framework for data-flow analysis, and explained how data-flow analysis can be viewed as a special case of constraint-based program analysis. We showed that cost analysis fits into constraint-based framework and focused on generating cost relations. Little attention was focused on solving the generated relations.
References


