Demand-Driven Interprocedural Analysis for Map-Based Abstract Domains

Kalmer Apinis, Varmo Vene, Vesal Vojdani

Institute of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia

Abstract

Many data flow analysis problems can be succinctly formalized using constraint systems. For interprocedural analysis, the system may contain an infinite number of constraints, but it can still be solved using a local solver that evaluates the constraints in a demand-driven fashion. In this paper, we use local solvers to develop a compositional framework for interprocedural on-demand static analysis. We can integrate any map-based abstract domain into our framework, such as a points-to analysis that maps pointers to their possible target addresses. Driven by the local solving algorithm that tracks the required dependencies, only those points-to sets that are of interest to the user are computed. The approach is applicable whenever the keys of the map are efficiently comparable and the domain operations are applied pointwise; we place no additional restrictions on the value domain nor on the transfer functions of the analysis beyond the standard termination requirements of the solvers.

Keywords: Static Program Analysis, Interprocedural Data Flow Analysis, Abstract Interpretation, Constraint Systems, Local Solving

1. Introduction

Consider the task of building a static program analyser to detect race conditions in Linux device drivers. One solution is to design an inter-procedural analyzer specific to this application [49, 34], but that makes it hard to extend and improve the precision of the analyzer. In contrast, the Goblint data race analyzer [48] is designed in a modular fashion, consisting of many smaller analyses and a query system that allows these subanalyses to benefit from one another. Auxiliary analyses, such as heap region analysis [40] and address equality analysis [42], can be developed separately and plugged into the inter-procedural framework to improve the precision of the tool. More generally, Rival et al. [37]
present a position paper advocating the compositional approach for building extensible static analyzers.

While the handling of pointer variables is critical for many analyses of real world C programs [47], the complete results of such support analyses are not particularly interesting to users who primarily care about their specific properties, e.g., data-race warnings. Therefore, it would be ideal if such unneeded information were not even calculated. Our proposal is to apply the method of demand-driven computation that is already used in static analysis frameworks: local solving [13]. This method consists of, first, setting up a constraint system, and then, solving the system using a local solver, e.g., the topdown solver by Le Charlier and Van Hentenryck [28] or Kildall’s worklist algorithm [25, 33]. These solvers do not necessarily compute the solution for all constraint system variables, but only for a set of chosen variables and variables that are needed during their computation.

For the data-race analysis example, we may set up the constraint system in such a way that different (sub-)analyses use separate constraint system variables—which may be then skipped by the local solver. The question is then if the idea can be used in conjunction with interprocedural program analysis. More generally, the goal of this paper is to investigate the details of making constraint-system–based analyses more demand-driven using local solvers. We aim to express demand-driven analysis by only changing the constraint system, so that we benefit instantly from recent (and future) algorithmic advances in constraint solving; in particular, local solvers for intertwined application of widenings and narrowings [1] already allows us to use more expressive numeric domains than possible in related frameworks for demand-driven analysis.

We propose a generic mechanism for cases where the analysis domain is a map structure (i.e., an associative array) such that only values for those keys will be computed which are needed. For this mechanism any analysis domain can be used as long as it can be expressed as a map where the least upper bound operation is pointwise. We will look at two specific abstract domains to exemplify the proposed mechanism—a map domain that combines different analyses, and points-to analysis that maps variables to their points-to sets. This idea, as we will see, only has the potential for improvement in the call string approach to interprocedural static program analysis. In the functional approach, however, all map keys will be needed for analyzing function calls.

We have a proof-of-concept implementation of this idea based on the Goblint analysis framework [3].

The structure of the paper. After the introduction, in Section 2, we will formalize some basic terms: what are programs and how we reason about them. In Section 3, we present the underlying idea of the paper: distributing map domains to separate constraint system variables. We start with the example of distributing helper analyses into separate constraint system variables. In Section 4, we look at interprocedural program analysis and how it affects our approach to demand-driven analysis by the example of points-to analysis. We look at the two classical approaches: the functional approach and the call string.
int g(int y) {
  return y + 1;
}

int main() {
  int x = 0;
  while (x<10)
    x = g(x);
  return x;
}

Figure 1: Counting program

approach. In Section 5, we extend the semantics with support for multithreaded
programs and present the distributing mechanism in that setting. The next sec-
tion contains the results of practical experimentation where we modified the
 analyzer Goblint. In Section 7, we briefly summarize related work; and in the
last part conclude the paper.

2. Program analysis

2.1. Programs

A program, for this paper, consists of a set of functions $\mathcal{F}$, of which one is
the main function $\text{main} \in \mathcal{F}$. Each function $f \in \mathcal{F}$ is represented by a control-
flow graph $(N_f, E_f, s_f, r_f)$ where $N_f$ is the set of program points, $E_f \subseteq N_f \times L \times N_f$ is the set of labeled edges, $s_f$ and $r_f$ are the start and return points
of the function $f$. There are three distinct kinds of edge labels: statements $s$, conditional guards $\text{Pos}(\ldots)$ and $\text{Neg}(\ldots)$, which always come in pairs, and
function calls $x = h(\ldots)$. We assume that program points of different functions
are disjoint and that each program point $x \in N_f$ has a path from the start point
$s_f$ in addition to a path to the return point $r_f$.

Example. An example of a program together with its source code is given in
Figure 1. The program consists of two functions: the main function and the
function $g$.

2.2. Constraint systems

In static program analysis, a constraint system is a set of inequations $x \geq f_x$
over some partial order $(\geq) \subseteq D \times D$ where the left-hand sides are variables
from some set $\text{Var}$ and the right-hand sides are expressions that may refer to
variables from that set $\text{Var}$. A variable assignment $\sigma \in \text{Var} \rightarrow D$ is a solution of
that constraint system if the relation holds in case the variables are substituted
with their values in $\sigma$. For static program analysis it is suggested that the
domain of the constraint system $D$ should have the structure of a complete
lattice such that it encodes logical statements in the implication order [7]. In
practice, partial orders with weaker requirements may be used as long as they provide a binary upper bound operation [6].

Given that all constraints on any variable \( x \) can be combined into a single constraint on that variable \( x \), the whole system can also be represented by a function \( f \in (\text{Var} \rightarrow D) \rightarrow \text{Var} \rightarrow D \) that maps variables of \( \text{Var} \) and variable assignments \( \text{Var} \rightarrow D \) to values \( D \). In that case a fixpoint of \( f \), \( \sigma = f \sigma \) is a solution for the constraint system.

There exists a trivial solution, in the case that \( D \) is a complete lattice, that maps all variables to the largest element \( \top \in D \) of the lattice. Such a trivial solution, though, is useless in argumentation about the program. For each constraint system variable we are interested in, it is more useful to have stronger logical statements, i.e., smaller lattice elements. Practically, computing a solution for a constraint system can be handed off to a solving algorithm that tries to find a non-trivial solution—several of such solvers have been proposed over the years [28, 43, 15].

**Example.** Consider the following constraint system with variables \( \text{Var} = \{X, Y, Z\} \) where the domain \( D = 2^\mathbb{N} \) contains subsets of natural numbers ordered by inclusion.

\[
X \supseteq \{1\} \quad Y \supseteq X \cup Z \quad Z \supseteq \{y + 2 \mid y \in Y \land y < 5\}
\]

The elementwise least solution to this system is \( X = \{1\} \), \( Y = \{1, 3, 5\} \), and \( Z = \{3, 5\} \).

### 2.3. Semantics

The semantics of programs can be composed from the semantics of statements, conditional guards, and function calls [43]. The intuition for the intra-procedural part of the semantics is that the execution of a statement (or a guard) \( l \) is a transformation of sets of input states \( x \subseteq S \) to a set of output states \( y \subseteq S \), i.e., \( l \in 2^S \rightarrow 2^S \), where \( l \) is the statement or guard. Each transformation is expected to be distributive over set union. Then, using these transformations the *meet over all paths* [24] solution can theoretically be computed using the control-flow graph.

An empty set of output states signifies that the execution does not reach past that statement, a singleton set tells us the only possible outcome of the execution, and a larger set can be used for signifying non-determinism or that the program point is reached several times. The specifics of the semantics are intentionally left open, such that many languages and systems would fit in this framework.

**Example.** To gather intuition, we briefly look at one interpretation of the example program in Figure 1. At the starting point of the main function the value of variable \( x \) is undefined, thus, the set of possible states \( d_{\text{start}} \subseteq S \) contains a state for all possible values of \( x \). As there are no other variables besides \( x \) in the main function, the possible set of states collapses to a singleton set after setting
x to zero, i.e., \([x = 0]\) \(d_{\text{start}} = \{s\}\). In the resulting state \(s\) the value of variable \(x\) is zero. Thus, \([\text{Neg}(x = 0)]\) \(\{s\}\) = \(\emptyset\) and \([\text{Pos}(x = 0)]\) \(\{s\}\) = \(\{s\}\).

Defining the concrete semantics of function calls requires care such that only valid inter-procedural paths are used [26]—a function call must not have an effect on the return sites of other calls. The semantics of parameter passing are specified by transformation functions for entering the called function \(\text{enter}_l\) and returning back to the caller \(\text{comb}_l\). The execution of a function consists of an \(\text{enter}_l\) transformation followed by the semantics of the function body, followed again by a \(\text{comb}_l\) transformation—where \(l\) is again the label of the function call edge in question.

Example. We continue with the example program in Figure 1, where we are about to consider the function call \(x = g(x)\) with the state \(s\) where variable \(x\) is zero. Then, entering the function we again get a singleton state \(\text{enter}_{x = g(x)} \{s\} = \{s'\}\), where variable \(y\) has the value zero. The next step is returning the value of \(y\) plus one. Thus, \([\text{return } y + 1] \{s'\} = \{s''\}\) where the return value is one in the state \(s''\). Returning from the function call, we use \(\text{comb} \{s\} \{s''\} = \{s''''\}\), that stores the return value one into the variable \(x\) in state \(s''''\). If the main function had other local variables, their value would be copied into \(s''''\) from the pre-call state \(s\).

Following Müller-Olm and Seidl [31], we construct a constraint system \(\mathcal{R}\) whose solution maps program points to the set of states that reach that point. In order to distinguish data flowing into distinct calls of the same function, we first build a constraint system \(\mathcal{S}\) whose least solution represents the semantics of so-called same-level runs; this captures the semantics of each function in isolation. For \(\mathcal{S}\), each program point \(n \in N_f\) (of the function \(f\)) would contain the transformation function such that given an entry state for the function \(f\) it produces the set of states reaching \(n\) from the given entry state where all recursive calls from the function \(f\) have returned. The domain of the system will be the lattice of functions from input states \(S\) to the set of output states \(2^S\). Then, for each edge \((u, l, v) \in E_f\) of each function \(f\) we have:

\[
\mathcal{S}[s_f] \supseteq \lambda x. \{x\} \\
\mathcal{S}[v] \supseteq \lambda x. [[l]([\mathcal{S}[u] x)] \quad \text{if } l \text{ is a statement or guard} \\
\mathcal{S}[v] \supseteq \lambda x. \bigcup_{d \in \mathcal{S}[u] x} \mathcal{comb}_l \{d\} (\mathcal{S}[r_f] d') \quad \text{if } l \text{ is a function call to } f
\]  

(S)

The same-level run for the start node of program function \(f\) is just the function that returns a singleton set containing its input state. In the case that the label between nodes \(u\) and \(v\) is a statement or guard, we take the transformation for the node \(u\) and compose it with the semantics of the label \(l\). Function calls, however, are more involved. The semantics of the function call consists of the transformation to enter the body of the function, the semantics of the body of the function, and the transformation \(\text{comb}\) that generates function call
semantics by applying function body semantics. The constraint considers each pre-call state separately and applies set union at the end.

With the above constraints capturing the abstract effect of each function, we now define the constraint system that describes the semantics of the programs as the sets of reachable states at each program point. Similarly to the previous constraint system, the constraints of $\mathcal{R}$ are defined according to each edge $(u, l, v) \in E_f$ for each program function $f \in F$:

$$
\begin{align*}
\mathcal{R}_{s_{\text{main}}} & \supseteq d_{\text{start}} \\
\mathcal{R}_v & \supseteq [l] \; \mathcal{R}_u & \text{if } l \text{ is a statement or guard} \\
\mathcal{R}_{s_f} & \supseteq \text{enter}_f \; \mathcal{R}_u & \text{if } l \text{ is a function call to } f \\
\mathcal{R}_v & \supseteq \bigcup_{d \in \mathcal{R}_u} \text{comb}_f \{d\} \; (\mathcal{S}_{s_f} \, d) & \text{if } l \text{ is a function call to } f 
\end{align*}
$$

In addition to operators $[l]$, $\text{enter}$, and $\text{comb}$ the constraint system is defined using $d_{\text{start}}$ that is set of starting states of the main function as well as the least solution of the same-level runs constraint system $\mathcal{S}$.

The first constraint is stating that the starting point of the main function will need to contain the starting states. The second constraint deals with the case where the label is a statement or guard. In those cases we just apply the semantics of the label to the states reaching program point $u$. The last two constraints deal with function calls, of which the first makes sure that the function start point is reached with the states coming from each call site. Likewise, the last constraint makes sure that the returning state is propagated back to the call site. Note that the latter is done using the transformation of the same-level runs from $\mathcal{S}$—using return states from $\mathcal{R}$ would be unsound as different calls to the same function would get mixed up.

2.4. Abstract interpretation

In theory, to verify that a program satisfies some safety property we would, first, compute the least (according to subset inclusion) solution to the constraint system $\mathcal{R}$ and, subsequently, check the property for each program point. However, in practice this approach fails already at the first step as the sets of states to be computed become too large for non-trivial programs.

To solve the computability problem, Cousot and Cousot [8, 7] proposed a technique called abstract interpretation where complete lattice elements represent sets of states. The method allows us to trade off precision for analysis speed by computing in a lattice with a limited number of elements.

More formally, we require a complete lattice $(\mathcal{D}, \subseteq)$ and a concretization function $\gamma \in \mathcal{D} \rightarrow 2^S$ which maps a lattice element to the set of states it represents. The ordering of the lattice must be subset inclusion of represented sets:

$$
x \subseteq y \iff \gamma(x) \subseteq \gamma(y) .
$$

Further, the greatest element must map to the full set $S$ and the least element must map to the empty set of program states $\emptyset$. To construct the constraint
system we need abstract versions of the semantic functions: \( \llbracket \ell \rrbracket \in D \to D \), \( \text{enter}^\ell \in D \to D \), and \( \text{comb}^\ell \in D \to D \to D \). These functions must over-approximate their concrete versions, i.e., for all \( d, d_1, d_2 \in D \):

\[
\llbracket \ell \rrbracket (\gamma(d)) \subseteq \gamma(\llbracket \ell \rrbracket^\ast d) \\
\text{enter}^\ell (\gamma(d)) \subseteq \gamma(\text{enter}^\ell d) \\
\text{comb}^\ell (\gamma(d_1), \gamma(d_2)) \subseteq \gamma(\text{comb}^\ell d_1 d_2)
\]

**Example.** A simple constraint system \( \mathcal{A} \) that over-approximates \( \mathcal{R} \) can be constructed based on the structure of \( \mathcal{R} \). For each \( (u, l, v) \in E \), we have the constraints:

\[
\begin{align*}
\mathcal{A}[s_{\text{main}}] &\supseteq d_{\text{start}}^\ell \\
\mathcal{A}[v] &\supseteq \llbracket \ell \rrbracket^\ast \mathcal{A}[u] & \text{if } l \text{ is a statement or guard} \\
\mathcal{A}[s_f] &\supseteq \text{enter}^\ell \mathcal{A}[u] & \text{if } l \text{ is a function call to } f \\
\mathcal{A}[v] &\supseteq \text{comb}^\ell \mathcal{A}[u], \mathcal{A}[r_f] & \text{if } l \text{ is a function call to } f
\end{align*}
\]

The first constraint makes sure that the start point of the main function contains the value \( d_{\text{start}}^\ell \)—which subsumes all states of the set \( d_{\text{start}} \). The following constraints are generated for each edge of the control-flow graph. If a label of the edge is a statement or a guard, the second constraint applies, and in case the label is a function call, the last two constraints apply. In the case of a statement or guard, the abstract transfer function must be applied to the value at \( u \) to get the value at \( v \). In the case of a function call, the program point at the start of the called function must be constrained with the value from the caller by applying the \( \text{enter}^\ell \) function. Finally, the program point after the call must be constrained by applying \( \text{comb}^\ell \) to the value before the call and to the value from the end of the called function.

The downside of the strategy in the previous example is that the states of different function calls get mixed together and thus produce spurious flow of information from one function call site to another call site of the same function. There are two classical approaches [45] to abstract interpretation that improve the precision in such cases: the *functional approach* and the *call string approach*. Both approaches can be expressed using constraint systems. The resulting systems, however, may use infinitely many variables, as there may be infinitely many representations of contexts. Thus, the solution of such a system is not computable. Such systems, however, can still be partially solved with a local solver in a demand-driven fashion, starting from a suitable set of constraint system variables [13]. Thus, before considering the two classical approaches, we first review some background on local solving.

### 3. Demand driven analysis

#### 3.1. Local solving

Local solvers, as mentioned in the introduction, do not necessarily give the solution for all variables, but only for a set of chosen variables and variables
that their computation depends on. Thus, if you are not specifically requesting the solution of a variable and it is not referenced during the solving process then the constraints of that variable will not be evaluated.

Example. The following constraint system (from [4]) over natural numbers

\[ y_{2n} \geq \max(y_{2n}, n) \quad y_{2n+1} \geq y_{6n+4} \]

uses infinitely many constraint system variables, but has at least one finite partial solution \( \{y_1 \mapsto 2, y_2 \mapsto 2, y_4 \mapsto 2\} \) which would be found by a local solver when queried with the value of \( y_1 \).

3.2. Multiple interconnected analyses

One way to formalize the combination of lattices \( D_1, D_2, \ldots, D_n \) is by using a Cartesian product \( D^* = D_1 \times D_2 \times \ldots \times D_n \). The simplest combination of abstract transfer functions, known as the direct product [10], can be defined as follows (numbers in subscript denotes analysis numbers):

\[
\begin{align*}
\llbracket d \rrbracket^j(d_1, \ldots, d_n) & : = (\llbracket d \rrbracket^1_1 d_1, \ldots, \llbracket d \rrbracket^1_n d_n) \\
\text{enter}^j(d_1, \ldots, d_n) & : = (\text{enter}^1_1 d_1, \ldots, \text{enter}^1_n d_n) \\
\text{comb}^j((k_1, \ldots, k_n), (d_1, \ldots, d_n)) & : = (\text{comb}^1_{1,1} k_1 d_1, \ldots, \text{comb}^1_{n,n} k_n d_n)
\end{align*}
\]

This construction, however, is not satisfactory as there is no interaction between the components. Nothing is gained by performing all analyses together. Immediately, Cousot and Cousot [10] proposed the idea of reduced product, that solves the problem with maximal precision—at least in theory. The reduced product domain, however, invokes the (non-computable) concretization function inside of abstract semantic computation thus making the implementation impractical. Another idea was provided by Granger [18]. Instead of a single function \( \text{reduce} \in D^* \rightarrow D \), each analysis has to provide a function \( \text{refine}_i \in D^* \rightarrow D_i \) such that for all \( i \leq n \), \( \text{refine}_i d \subseteq d_i \). The \( \text{refine} \) functions would be used in a decreasing iteration phase that is performed after applying abstract semantics functions.

In Granger’s approach, though, communication between analyses only happens in the refinement phase. Another approach would be to allow direct read access to all analyses by passing the whole product to each analysis’s abstract semantic function: \( \llbracket d \rrbracket^j \in D^* \rightarrow D_j \) for each analysis \( j \). This method is straightforward but giving the whole \( n \)-tuple as an argument obscures the fact that the application of the transfer function of one subanalysis does not necessarily need pre-states of all other analyses. Such subanalyses typically depend only on the pre-state of their own lattice and, additionally, states of a small set of other analyses.

Conceptually, we could evaluate transfer functions in a lazy fashion, but all local solvers, as a consequence of needing to tabulate constraint system variables with their values, require complete evaluation of right-hand sides. Thus,
if we intend to rely on independent local solvers, we need to explicitly reformulate
the Cartesian product lattice such that the solving engine will not trigger
unnecessary computations.

Therefore, in this paper we are going to formalize the combination of lattices
as mappings $\mathbb{N} \rightarrow D^\cup$ where $D^\cup$ is the distinct union of the analyses lattices $D_1, D_2, \ldots, D_n$. Note that this is isomorphic to the Cartesian product lattice.
Each program point will be associated with a value $d \in \mathbb{N} \rightarrow D^\cup$ so that $d_i \in D_i$ is the state of the $i$-th subanalysis (for $i \leq n$). The transfer functions are
given by $[l]^d_i \in (\mathbb{N} \rightarrow D^\cup) \rightarrow D_i$, $enter^d_i \in (\mathbb{N} \rightarrow D^\cup) \rightarrow D_i$, $comb^d_i \in (\mathbb{N} \rightarrow D^\cup) \rightarrow (\mathbb{N} \rightarrow D^\cup) \rightarrow D_i$ such that they can be combined into abstract transfer
functions for the $D = \mathbb{N} \rightarrow D^\cup$ domain:

$$
[l]^d_i := [l]^d_i \quad enter^d_i := enter^d_i \quad comb^d_i d_1 d_2 i := comb^d_i d_1 d_2
$$

*Example.* Let us analyze the program in Figure 2 to find out which locks pro-
tect each (global) variable. We use a must-lockset analysis (1) together with a
points-to analysis (2). Notice that the lockset analysis requires points-to data to
dereference the variable $m$. At the beginning of main the lockset is empty and
we have no information about pointers. In the first call to `munge` `v` will point
to the variable $x$ and $m$ will point to the lock $m1$. Thus, the lockset analysis
will add $m1$ to the set of definitely held locks. During the access of the variable
$x$, lock $m1$ is definitely held. Similarly, in the next calls to `munge` the variable $y$
will be protected by $m2$ and $z$ will be protected by $m2$. The constraint system $A$, however, will merge points-to information of all calls to `munge` and therefore
will be unable to give a singleton points-to set to the parameter $m$, thus, the set
definedly held locks would remain empty during the access of $v$. 
3.3. Currying and un-Currying transformations

Applying multiple analyses with $D = \mathbb{N} \rightarrow D^\cup$ to the constraint system $A$, described by the function $f \in (\text{Var} \rightarrow D) \rightarrow \text{Var} \rightarrow D$, gives us

$$f \in (V \rightarrow \mathbb{N} \rightarrow D^\cup) \rightarrow V \rightarrow \mathbb{N} \rightarrow D^\cup$$

where $\text{Var} = V$ and $D = \mathbb{N} \rightarrow D^\cup$. We note that un-Currying [5] could be applied here to transform the constraint system into

$$f' \in (V \times \mathbb{N} \rightarrow D^\cup) \rightarrow V \times \mathbb{N} \rightarrow D^\cup$$

where $\text{Var} = V \times \mathbb{N}$ and $D = D^\cup$ instead. More formally, the new constraint system $A'$ can be defined using the old system with $f'(\sigma', x, n) := f \sigma x n$ where $\sigma x n := \sigma'(x, n)$ for each $\sigma' \in (V \times \mathbb{N}) \rightarrow D^\cup$, $x \in V$, and $n \in \mathbb{N}$. It is easy to see that the equation could be reversed, and, as the constraint system $A'$ is derived from $A$ by simply un-Currying. Generalized, we can state this as the following lemma.

**Lemma 1.** Given the sets $V, X$ and the complete lattice $(D', \sqsubseteq')$. Given, furthermore, two functions $f \in (V \rightarrow X \rightarrow D') \rightarrow V \rightarrow X \rightarrow D'$ and $f' \in (V \times X \rightarrow D') \rightarrow V \times X \rightarrow D'$ which represent constraint systems $A$ and $A'$, respectively. Assuming that for all $v \in V$ and $x \in X$, $f \sigma v x = f' \sigma' v x$ where $\sigma$ and $\sigma'$ are functions such that $\sigma v x = \sigma'(v, x)$. Then, $\sigma$ is a solution of $A$ if and only if $\sigma'$ is a solution of $A'$.

**Proof.** Follows from the fact that Currying and un-Currying are each others inverse functions. \qed

By querying the constraint system $A'$ for the variable $(r_{\text{main}}, n)$ for some analysis $n$, the solver has to compute other analyses only if they are (transitively) needed. Thus, adding an analysis to the system has no computational impact in the case that it is not needed for the specific analysis run for a fixed program. Moreover, the dependence on some analysis need not be binary. If an additional analysis is needed for some query about the program, it is not necessarily performed for the entire program; indeed, whenever the result of any subanalysis is not needed for some program point, the evaluation of the right-hand sides of the related system variables is never triggered.

**Example.** Imagine that the munge function (in Figure 4) is inlined into the main function and then analyzed in same setting as earlier but now with the constraint system $A'$. Then, the whole points-to analysis is effectively skipped as it is not queried by the lockset analysis.

We demonstrated that the procedure of un-Currying can be applied to any mapping $D = X \rightarrow D'$ to distribute elements of $X$ to different constraint system variables. One more source of maps arises in program analysis when some information is stored for each program variable $V$, e.g., constant propagation or points-to analysis. To get a better idea of how distribution to separate variables may affects the interprocedural analysis process, we are going to examine the case of distributing a points-to analysis to separate variables. However, we will first review the two classical approaches to interprocedural data flow analysis.

10
4. Interprocedural Analysis

The constraint system $A$ may result in precision loss if a function is called in several places in the program. This happens because the states at the return point of the function are propagated to all its call sites. Exploiting knowledge about interprocedural flow, Sharir and Pnueli [45] proposed two approaches to increase analysis precision with respect to function calls: the functional approach and the call-string approach.

4.1. The Functional Approach

For all $(u, l, v) \in E$ and all values $d \in D$ we have the following constraints:

\[
\begin{align*}
\mathcal{I}[s_f, d] & \sqsubseteq d \\
\mathcal{I}[u, d] & \sqsubseteq \text{cond} \mathcal{I}[u, d] & \text{if } l \text{ is a statement or guard} \quad (I) \\
\mathcal{I}[u, d] & \sqsubseteq \text{comb}^\uparrow \mathcal{I}[u, d], \mathcal{I}[r_f, \text{enter}^\uparrow \mathcal{I}[u, d]] & \text{if } l \text{ is a function call to } f
\end{align*}
\]

A variable $\mathcal{I}[v, d]$ of the constraint system $\mathcal{I}$ consists of a program point $v \in N_f$ of some function $f$ and a lattice element $d \in D$. The lattice element corresponds to the entry state of the function $f$ at hand. Therefore, as the first constraint, the value of the starting point of any function $f$ is the argument of the function $d$. The second constraint deals with statements or guards by applying the transfer function to the value of the previous node. The third and last constraint handles the function calls, by applying the $\text{comb}^\uparrow$ operator to the value before the call and the value coming back from the call. Note that here the lattice element $\text{enter}^\uparrow \mathcal{I}[u, d] \in D$ is taken as the context of the call to $f$. We see that invalid interprocedural paths are excluded as the values at the return points of function calls are propagated to call sites only for matching entry states.

Because the total number of solver variables in the constraint system $\mathcal{I}$ is very large or even infinite, computing the whole solution is impractical at best. We can, however, use the local solving approach described in Section 3.1. This returns a partial solution $\sigma$ such that for each constraint system variable $x \in \text{dom}(\sigma)$, the right-hand sides of $x$ can be evaluated using values in $\sigma$ and, moreover, the constraint for $x$ is satisfied. Thus, to find an over-approximation of the concrete semantics, it suffices to use local solving to find the value of the constraint system variable $\mathcal{I}[r_{\text{main}}, d_{\text{main}}]$, where $d_{\text{main}}$ is the sound over-approximation of the stating state. The resulting partial solution, by construction, contains values for all reachable program points for all reachable contexts such that all relevant constraints are satisfied.

\textit{Example.} Let us use the constraint system $\mathcal{I}$ to analyze the program in Figure 1 using the constant propagation domain that maps variables to either a constant value or $\top$ if the value is not provably a constant. The analysis starts by considering the constraint system variable $\mathcal{I}[r_{\text{main}}, d_{\text{main}}]$, but as the transfer functions depend on their input, we again start transforming the state from the beginning of $\text{main}$. The entry state and the context of the main function is
{x \mapsto \top}. After the first statement we get \([x = 0]\{x \mapsto \top\} = {x \mapsto 0}\). As the constant 0 is smaller than 10 we get \([\text{Pos}(x < 10)]\{x \mapsto 0\} = {x \mapsto 0}\). Now we call the function \(g\) with the entry state (and context) \(\text{enter}^2\{x \mapsto 0\} = \{y \mapsto 0\}\). The return value of \(g\) will be \([\text{return} y+1]\{y \mapsto 0\} = \{R \mapsto 1\}\). This will be combined in the main function as \(\text{comb}^2\{x \mapsto 0\} \{R \mapsto 1\} = \{x \mapsto 1\}\). As this call looped back to the conditional jump, the solver joins the abstract values into \({x \mapsto 0}\) \(\cup \{x \mapsto 1\} = \{x \mapsto \top\}\). The next (and final) iteration of the loop calls the function with state \({x \mapsto \top}\) (and with \{y \mapsto \top\} inside \(g\)). We note that in the final solution the return value of \(g\) in the context \{y \mapsto 0\} is \{R \mapsto 1\}, whereas for the context \{y \mapsto \top\}, it is \{R \mapsto \top\}—the latter does not overwrite the former.

We can instantiate the constraint system \(\mathcal{I}\) with demand driven points-to analysis where the domain \(D\) is a mapping from program variables \(V\) to sets of variables that includes the null pointer \(2^V+1\). Let us denote that Curried constraint system by \(\mathcal{I}'\) whose variables will now be triples \(\mathcal{I}[n, c, a] \in \text{Var}\) that contain the program point \(n \in N_f\), for some function \(f \in F\), the abstract state \(c \in D\) at the entry point of \(f\), and program variable \(a \in V\). The notation \(\mathcal{I}[n, c]\) is meant to signify a partial application of the variable—equivalent to the lambda–expression \((\lambda x. \mathcal{I}[n, c, x])\). The constraint system \(\mathcal{I}'\) is thus:

\[
\begin{align*}
\mathcal{I}[s_g, c, a] & \supseteq c \ a & g \in F \\
\mathcal{I}[v, c, a] & \supseteq [v] \mathcal{I}[u, c] & (u, l, v) \in E \\
\mathcal{I}[v, c, a] & \supseteq \text{comb}^2 \mathcal{I}[u, c] \mathcal{I}[r_g, \text{enter}^2 \mathcal{I}[u, c]] & (u, g(), v) \in E.
\end{align*}
\]

The first constraint looks up the value of the variable from the context. The second constraint is a bit more complicated as we pass the partially applied function \(\mathcal{I}[u, c]\) to the transfer function—but this only means that the transfer function can query the values of \(\mathcal{I}[u, c, b]\) for any variable \(b\). The last constraint, however, is even more complicated: notice the partially applied function \(\text{enter}^2 \mathcal{I}[u, c]\) that is used as the context for querying the abstract value at the return point \(r_g\). For the solver to decide if the variable \(\mathcal{I}[r_g, \text{enter}^2 \mathcal{I}[u, c]]\) is new or already considered, it needs to tabulate the functional value \(\text{enter}^2 \mathcal{I}[u, c]\) in the context. Thus forcing analysis of all program variables at each call site.

What prevents us from expressing fine-grained demand-driven analysis as a constraint system is fundamental to the working of the functional approach. Imagine, in a C-like language, a Boolean variable \(\text{mode}\). Analysis of calls to some function \(f\) with \(\text{mode} = \text{true}\) and \(\text{mode} = \text{false}\) have by design separate context in the functional approach regardless of whether \(\text{mode}\) is referenced in \(f\). Thus, the demand-driven analysis would have to calculate the value for \(\text{mode}\) even if its value is not read by the transfer functions. We conclude that using the proposed mechanism with the functional approach does not lead existing local solvers to avoid unneeded computation.

### 4.2. The (Abstract) Call-String Approach

Where the functional approach differentiates between different calls to some function \(f\) based on the abstract value at the start of the function, the call
string approach differentiates calls by their respective states of the call stack. This is achieved by inserting a representation of a call stack state, i.e., the call string, into the variable of the constraint system domain—instead of the starting state of the function. This has the result that calls from different locations are kept separate and solved independently even if the abstract states at the start of the function are equal.

In the case that the program is recursive, it can happen that the set of call strings that occur during the analysis is infinite. For such cases *abstract call strings* can be used to limit the number of possible call string values. The mathematical structure \((\text{CS}, \varepsilon, ::)\) is an abstract call string if \(\text{CS}\) is a set, \(\varepsilon \in \text{CS}\), and the infix operator \(::\) maps an edge to an abstract call string. This means that we are free to choose a structure where \(\text{CS}\) is finite and, thus, also \(\text{Var}\) is finite. For all edges \(e = (u, l, v) \in E\) and call strings \(c \in \text{CS}\) we have

\[
\begin{align*}
C[\text{sf}, e :: c] & \equiv \text{enter} f C[u, c] & \text{if } l \text{ is a function call to } f \\
C[v, c] & \equiv \text{comb} f C[u, c] C[r_f, e :: c] & \text{if } l \text{ is a function call to } f
\end{align*}
\]

The first constraint sets the starting point of the main function with an empty call string to the starting value \(d_{\text{start}}\). The second constraint of \(C\) takes care of statement and guard edges by applying the appropriate transfer function. Note that the same call string \(c\) is used in both sides of the constraint—as statements and guards do not change the call string. The last two constraints handle function calls. The third constraint applies the \(\text{enter} f\) function and appends the current edge \(e\) to the call string. The last constraint applies \(\text{comb} f\) and restores the call string state to the one before the call.

Note that we can take a singleton set \(\{\varepsilon\}\) as the set of call strings. In that case, no actual differentiation is done and the resulting constraint system is isomorphic to \(A\).

Now, let us return to our demand-driven points-to analysis. Inlining the abstract semantics functions of the points-to analysis into the constraint system \(C\) produces the following constraint system:

\[
\begin{align*}
C[\text{sfmain}, \varepsilon, a] & \equiv d_{\text{start}}^a \\
C[s_g, c, a] & \equiv \text{enter} g C[u, c'] a & (u, g(), v) \in E, c = (u, v) :: c' \\
C[v, c, a] & \equiv [[l]^2 C[u, c] a & (u, l, v) \in E \\
C[v, c, a] & \equiv \text{comb} g C[u, c] C[r_g, (u, v) :: c] a & (u, g(), v) \in E.
\end{align*}
\]

The first two constraints take care of the cases of entering a function call based on whether the abstract call string is empty and we are in the main function, or the call string is not empty. The second constraint, in effect, requires us to decompose the call-string \(c\) into the call site \((u, v)\) and call string \(c'\). In general, there is no reason to assume that the operator \(::\) has an inverse.
Note, however, that we only decompose call-strings that we compose in the fourth constraint, so we can store the needed decomposition during composition in some mutable data-structure. Thus for the second constraint, if we know the program point before the call \( u \) and the call string of the call site \( c' \), then we can computing the entry state for variable \( a \). Furthermore, the function \( C'[u, c'] \) is passed as the argument to \( \text{enter}^4 \)—it can be used to query different values \( C'[u, c', b] \).

The next constraint deals with non-call edges. Although the function \( C'[u, c] \) is passed as the argument to \( [\ell] \), it is again only used to query different values \( C'[u, c, b] \). For example, the function \( \text{comb} \) may decide if the value of variable \( a \) comes from the called function \( C'[r_y, (u,v) :: c,a] \), from before the call \( C'[u, c, a] \), or some combination of the two.

**Example.** Let us perform points-to and value analysis for the program in Figure 3 using \( C'' \) (Currying transformation done twice). For each constraint system variable \( C''[u, c, a, x] \), \( u \) is the program point, \( c \) the abstract call string, \( a \) the analysis number, and \( x \) the program variable. Let us denote points-to analysis with 0 and value analysis with 1. The constraints needed to compute the return variable of the main function are the following.

\[
C''[m4, \varepsilon, 1, R] \supseteq C''[m3, \varepsilon, 1, y] \\
C''[m3, \varepsilon, 1, y] \supseteq \text{if } y \in \{y\} \text{ then } C''[f2, f :: \varepsilon, 1, y] \text{ else } C''[m2, \varepsilon, 1, y] \\
C''[f2, f :: \varepsilon, 1, y] \supseteq \text{if } y \in C''[f1, f :: \varepsilon, 0, v] \text{ then } 20 \text{ else } C''[f1, f :: \varepsilon, 1, y] \\
C''[f1, f :: \varepsilon, 0, v] \supseteq \{y\}
\]

The analysis is finished after evaluating these four constraints and we do not need to look at rest of the code (shown as ellipses). Thus, in suitable settings, the approach can reduce the analysis time to a constant regardless of the program size.

These kinds of improvements, however, are only doable if we design the analyses with demand-driven operation in mind. The analysis presented in
this example skips reachability computation—in case program point m2 is not reached, nothing will be returned. The reachability check, however, would force the solver to look at most statements of the program.

We saw that unlike the functional approach the call-string approach is a syntactic construction where the dependencies on specific program variables are not inherently affected by the procedure call construction. Thus, the analysis designer may freely decide the condition when to trigger the calculation of a specific program variable.

5. Analysis of multithreaded programs

As we turn to multithreaded programs, the question is again whether we can obtain a demand-driven analysis by rewriting the constraint system, but now we adapt constraints for thread-modular inter-procedural analysis. Let us start by formalizing the concept of multithreaded programs. The program states \((l,g,w) \in S\) are separated into three components where \(l \in L\) is the thread-local portion of the active thread, \(g \in G\) is the globally accessible state, and \(w \in 2^{N \times L}\) is the set of inactive threads. An inactive thread, here, consists of its current location together with its local state at that location. Thus, spawning of new threads might be implemented as

\[
\text{\texttt{spawn(&f)}}(l,g,w) := \{(l,g,w \cup \{s_f,d\}) \mid d \in d_{\text{start}}\}.
\]

Spawning a new thread for the function \(f\) inserts the starting point of \(f\) into the set of inactive threads together with its fresh thread-local state. Now it suffices to extend the constraint system with context-switching where the current thread becomes inactive and a previously inactive thread becomes active. To achieve this, we add a constraint for each program point pair \(u, v \in N\) such that the execution will ‘jump’ from one thread at program point \(u\) to another thread at program point \(v\):

\[
R[v] \supseteq \{(l',g,(w \setminus \{(v,l')\}) \cup \{(u,l)\}) \mid (l,g,w) \in R[u], (v,l') \in w\} \quad (R_m)
\]

If at program point \(u\), the local state is \(l\) and there is an inactive thread at point \(v\) with local state \(l'\), then a context-switch can ‘jump’ to program point \(v\) and local state \(l'\)—thus we need to constrain \(R[v]\). The global state \(g\) will be unaffected by this but the set of inactive threads \(w\) is updated to reflect the context-switch.

A naive approach to extending the analysis with multithreading support would be to add similar constraints to our analysis constraint system. This, however, is not practical as there are too many possible thread interleavings. Assuming at least two threads are active, each live constraint system variable is a potential context-switch target of each other live constraint system variable from other threads.

A practical solution to address the interleaving issue was proposed by Seidl et al. [41]—handle global variables flow-insensitively using side-effecting constraint systems. Previously, we had constraint system variables \(C[u,c]\) that
over-approximated the program state at the control flow graph node \( u \) and some call string \( c \) (or context in case of the functional approach). Now we extend the system with constraint system variables \( C_g \), which over-approximate all values of the global program variable \( g \) at any program point.

As reasoned by Apinis et al. [3], we do not want to generate constraints for variables \( C_g \) as the number of potential modification sites of \( g \) is too large. Instead, the transfer function at the modification site should also perform the update to \( C_g \)—as a side-effect to the normal right-hand side evaluation. Side-effecting constraint systems allow each right-hand side of a constraint to have a contribution (side-effect) to another variable besides its left-hand side. This is achieved by adding a parameter \( \text{set} : \mathbb{V} \rightarrow \mathbb{D} \rightarrow \mathbb{1} \), where \( \mathbb{1} \) specifies the unit type, to the signature of the constraint function. By calling the function \( \text{set} \), the analysis can send a signal the environment about contributions to global variables. For symmetry, we name the first argument of the constraint system function \( \text{get} : \mathbb{V} \rightarrow \mathbb{D} \). Thus, the complete type of the side-effecting constraint systems function is

\[
f \in (\mathbb{V} \rightarrow \mathbb{D}) \rightarrow (\mathbb{V} \rightarrow \mathbb{D} \rightarrow \mathbb{1}) \rightarrow \mathbb{V} \rightarrow \mathbb{D}.
\]

A variable assignment \( \sigma : \mathbb{V} \rightarrow \mathbb{D} \) is a solution for the side-effecting constraint system given by the function \( f \) if for each variable \( x \in \mathbb{V} \) it holds that \( \sigma x \sqsubseteq f \sigma \text{set} \) such that for each call to \( \text{set} \ y \ d \) from inside \( f \) it also holds that \( \sigma y \sqsubseteq d \).

**Example.** Let us look at the following side-effecting constraint system with variables \( \mathbb{V} = \{a, b, g\} \) over the domain \( \mathbb{D} = \mathbb{N} \cup \{\infty\} \) of natural numbers extended with infinity.

\[
\begin{align*}
a & \sqsubseteq \text{let } \_ & = \text{set } g (\text{get } a + 1) \text{ in } 1 \\
b & \sqsubseteq \text{get } a + \text{get } g
\end{align*}
\]

The first constraint contributes the value 1 to \( a \), but before that it contributes the value of \( a + 1 \) to \( g \). The second constraint contributes the sum of \( a \) and \( g \) to the variable \( b \). The least solution is \( \{a \mapsto 1, g \mapsto 2, b \mapsto 3\} \).

To formalize our analysis as a side-effecting constraint system, we need to modify our transfer functions to have getter and setter functions that read and update global program variables \( G \). For the value-domain \( \mathbb{G} \) this would be done as follows.

\[
\begin{align*}
[l] & \in (\mathbb{G} \rightarrow \mathbb{G}) \rightarrow (\mathbb{G} \rightarrow \mathbb{G} \rightarrow \mathbb{1}) \rightarrow \mathbb{D} \rightarrow \mathbb{D} \\
\text{enter} & \in (\mathbb{G} \rightarrow \mathbb{G}) \rightarrow (\mathbb{G} \rightarrow \mathbb{G} \rightarrow \mathbb{1}) \rightarrow \mathbb{D} \rightarrow \mathbb{D} \\
\text{comb} & \in (\mathbb{G} \rightarrow \mathbb{G}) \rightarrow (\mathbb{G} \rightarrow \mathbb{G} \rightarrow \mathbb{1}) \rightarrow \mathbb{D} \rightarrow \mathbb{D} \rightarrow \mathbb{D}
\end{align*}
\]

In the programming language C, global variables are typically initialized by loading the executable binary into memory. As this step is not present explicitly in code, the main function will need to be updated to perform the
required initialization. Thus, assuming global variables are already initialized at the point where the starting thread spawns another thread, the constraint system can be written as

\[
\begin{align*}
C[\text{main}, e] & \supseteq d_{\text{start}}^e \\
C[v, c] & \supseteq [l]^d \text{ get set } C[u, c] & (u, l, v) \in E & (C_m) \\
C[s_f, e :: c] & \supseteq \text{ enter}_\text{f} \text{ get set } C[u, c] & (u, g(), v) \in E \\
C[v, c] & \supseteq \text{ comb}_\text{f} \text{ get set } C[u, c] C[r_f, e :: c] & (u, g(), v) \in E
\end{align*}
\]

In effect, we just allow the semantics functions to read and update values of global program variables. Note that this kind of handling of global variables is flow-insensitive. Local variables can have different values depending on the program point, whereas global variables have only one value. This is done by design so that we would not need to model thread interleavings [41].

Example. Let us analyze the program in Figure 4 to find out which locks protect each global variable. We use a must-lockset analysis (1) together with a points-to analysis (2). Notice that the lockset analysis requires points-to data to dereference the variable \( m \). At the beginning of \texttt{thread1\_main} the lockset is empty and we have no information about pointers. In the first call to \texttt{munge}, \( v \) will point to the global variable \( x \) and \( m \) will point to the lock \( m1 \). Thus, the lockset analysis will add \( m1 \) to the set of definitely held locks. During the access of the global variable \( x \), lock \( m1 \) is definitely held. Similarly, in the next calls to \texttt{munge} the global variable \( y \) will be protected by \( m2 \) and \( z \) will be protected by \( m2 \). Similar steps will be done in the second thread, but with the exception that the variable \( y \) is now protected by \( m2 \) instead. As the variable \( y \) is not consistently protected by any single lock, we cannot exclude race conditions based on locking.

Next we can combine the method of making the analysis more demand-driven from the previous section to the multithreaded call-string approach from this section. The resulting constraint system is the following:
Distributing a map domain to separate variables in the multithreaded setting poses a couple of practical challenges. First, we have to decide which map key (which right-hand side) will do the side-effects to globals. In the case of our experimental implementation of the points-to analysis we decided to artificially add a “local” program variable $F$ for two purposes: to compute in a single place the reachability property of the particular program point, and to perform side-effects. The second challenge is to only side-effect to global variables that are (transitively) needed. For this we added a “global” variable $G$ that, in the analysis, holds the set of all needed global variables. We present the pattern only for $\text{JlK}^\#$ as the pattern for functions $\text{enter}^\#$ and $\text{comb}^\#$ is analogous:

$$\text{JlK}^\# l\text{get gset get } F := \text{if get } F = \text{reachable and multithreaded then}$$

$$\text{for } g \in \text{gget } G \text{ do set } g ((\text{ll})^\# \text{gget gset get } g)$$

... 

Thus, for each kind of edge, the constraint for the $F$ variable will perform the side effects of all needed globals $g \in C'[G]$ if the program point is reachable and in multithreaded mode, finally computing reachability for the next program point. Note that the transfer functions are responsible for querying the flow-insensitive state in multithreaded mode and flow-sensitive variables in single-threaded mode.

6. Experimental evaluation

The example in Section 4.2 has established that large portions of the programs may possibly be skipped by the demand-driven analysis. Thus, we can now write specific analyses to exploit this, but looking at the best case performance increase is uninformative. Instead, we present our findings on how the proposed transformation in Subsection 3.3 behaves for analyses that were not designed with demand-driven operation in mind. In this case, we do not expect a speed increase out of the box because these analyses still process irrelevant right-hand sides, but the question is whether there is potential for saving, i.e., whether fewer program variables are evaluated in our analysis applications.

We implemented two constraint systems on top of the Goblint static analyzer framework which analyses programs written in the C language$^1$. Both new constraint systems use the call string approach to perform points-to and

---

$^1$Available from https://github.com/kalmera/analyzer
constant propagation analysis. All evaluation was done on an Mid-2014 Apple MacBook Pro laptop running OS X version 10.11.6. The first is the implementation of the constraint systems $C_m'$ that distributes the analysis domain between different constraint system variables. The second, serving as control, is the same analysis using constraint system $C_m$. The implementations were written to be equivalent to the constraint systems $C'$ and $C$, respectively, when analyzing single-threaded programs.

We note that the combined points-to and constant propagation analysis, using the functional approach, is used by Goblint to perform data-race analysis. The domain of the analysis in our new setting is a mapping whose keys is either a program variable or a special name $F$. The program variables are mapped to a lattice structure that contains points-to and constant-propagation lattices while $F$ maps to information about the current thread and reachability of the program point in general. The modifications needed for this paper were extensive, as, first, Goblint used the functional approach, and second, transfer functions had to be Curried manually. Only very light testing was performed and no optimization was done.

The first program that we analyzed is the mathematical simulation code 433.milc (su3imp) taken from SPEC CPU2006 benchmark suite [20]. The pre-processed code consists of 12,274 lines of C code which is converted into 7,165 control flow graph nodes. The program has 238 functions and 92 global variables.

Comparative results of the experimental evaluation for the first program are given in Table 1. Solving the combined constraint system reached the partial solution in about 2.3 minutes—giving values to 141,280 constraint system variables. As the analysis domain is a mapping, Goblint computed 16,896,112 values for program variables and $F$-s. This is our baseline to compare against as it has fully analyzed all program variables in all program points and reachable call strings. If we are really interested in values for all variables then the first constraint system and its solution are sufficient. Only if values for some variables were not required would we hope to gain any efficiency. Thus, for the second constraint system we will take a extreme stance and only compute the reachability of the return node of the main function. This, however, forces the computation of all program variables that are involved in control-flow as the main function is not necessarily assumed to even terminate. In addition, all arguments to API calls are computed as they might overwrite needed variables.

<table>
<thead>
<tr>
<th></th>
<th>Full ($C_m$)</th>
<th>Demand-driven ($C_m'$)</th>
</tr>
</thead>
<tbody>
<tr>
<td># of variables computed</td>
<td>16,896,112</td>
<td>5,322,855</td>
</tr>
<tr>
<td>% of variables computed</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>Time (min)</td>
<td>2.3</td>
<td>6.1</td>
</tr>
</tbody>
</table>

Table 1: Results for 433.milc (su3imp)
Table 2: Results for the Linux device driver lp

<table>
<thead>
<tr>
<th></th>
<th>Full ($C_m$)</th>
<th>Demand-driven ($C'_m$)</th>
</tr>
</thead>
<tbody>
<tr>
<td># of variables computed</td>
<td>24,253</td>
<td>9,677</td>
</tr>
<tr>
<td>% of variables computed</td>
<td>100</td>
<td>38</td>
</tr>
<tr>
<td>Time (s)</td>
<td>0.6</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Solving the second constraint system took about 6.1 minutes and resulted in the partial solution containing values for 5,322,855 constraint system variables. This means that we only needed to compute about 32% of variables, showing that it is possible to significantly reduce computational effort and memory footprint. The number of constraint variables increased about 2.6 times and analysis time increased about 2.7 times. The correlation of these numbers is not surprising as the implementation of transfer functions have similar overhead when comparing the un-Curried and the Curried versions. Therefore, as more constraint system variables are evaluate also more transfer functions are evaluated—increasing the total analysis time. In the current setting where reachability must be computed for every program point it is impossible for the distributed constraint system to compute fewer constraint variables than the combined constraint system. This need not necessarily be that way—the number of computed program variables would be significantly smaller if the analysis would assume that all program points were eventually reachable.

Next we analyze at the Linux device driver lp: the generic parallel printer driver. Inside the driver, 402 global variables are statically accessible together with 1,638 functions; where only 24 functions are defined in the lp.c file and the rest come from header files. All the functions together generated 9,263 nodes in the control-flow graphs.

Comparative results of the experimental evaluation for the second program are given in Table 2. Solving the combined constraint system for lp.c required computation of 2,138 constraint system variables which incorporated values for 24,253 program variables whereas solving the reachability using the distributed constraint system needed to solve 9,677. The combined constraint system being faster, taking only 0.6 seconds as solving the distributed constraint system took about 2.1 seconds. The overall result is similar to the single-threaded case: the new approach needed about 38% of values, computed 4.5 times more constraint system variables, and it took 3.5 times more time.

The experimental evaluation shows that distributing analyses to separate variables behaves as intended: allowing for fine-grained demand-driven computation. It is, however, up to the implementer of the analyses to make sure that the separation is done efficiently and that repeated calculation is avoided where possible.
7. Related work

Demand-driven analysis is particularly useful for assertion checking since an assertion may often be determined locally. Symbolic execution can be used to find counter-examples that violate assertions. Anand et al. [2] present a demand-driven approach to symbolic execution, while Manevich et al. [30] present a post-mortem static analysis that tracks the flow of a single value from the error location to the point in the program where it may have originated. Assertions can also be proven in Hoare logic, and static analyzers can be used to assist the process by inferring invariants and resolving alias queries [16]. For this, [44] present a goal-directed approach to slice static analysis results in order to make the proof obligations smaller. A number of specific demand-driven analyses have also been proposed that can be used to answer, e.g., aliasing queries [19, 46, 51, 17].

When it comes to general frameworks, demand-driven extensions have been proposed to the classic framework for IFDS (Interprocedural Finite Distributive Subset) problems [36]. Sagiv et al. [38] extended the framework to compute map-based domains in a demand-driven manner as long as the transfer functions are distributive. They consider copy-constant and linear-constant propagation and find a speed-up of 1.14 to 6 times for the demand-driven algorithm. Duesterwald et al. [12] present a modification based on reversing transfer functions [22]. Transfer functions can be automatically reversed for finite domains, such as that of copy-constant propagation, but it is not clear how to reverse general constant propagation. More recently, Naeem et al. [32] make a number of practical extensions to the IFDS algorithm, allowing among others demand-driven creation of the interprocedural super-graph. Another interesting approach is the automaton-based iterative algorithm of the Explorer tool [14], which can check properties expressed as regular expressions on the call stack configurations.

Instead of constraint systems and local solving, other formalisms could be used, e.g., minimal function graphs that can be used to analyzing a function for a given entry state [23]. The choice of formalism and backend solver, however, does not affect issues arising from context sensitive demand-driven analysis. Similarly, when analysis problems can be formulated as logic programs, efficient demand-driven datalog solving can be obtained using magic-set transformation [35, 39] and Binary Decision Diagrams [27, 50]. These methods are limited to abstract domains that can be expressed within datalog and cannot express infinite value domains. Madsen et al. [29] present a datalog-like language for computing fixpoints on arbitrary lattices. They can handle the same family of interprocedural analysis problems, IFDS and extensions, described in the previous paragraph.

In contrast to the above mentioned work, our approach allows the demand-driven solving of any map-based static analysis in an interprocedural and even multithreaded setting. This includes numeric abstract domains that require widening/narrowings, such as Interval Analysis [9]; however, it does not include relational domains such as Convex Polyhedra [11].
8. Conclusion

Problems of interprocedural static program analysis can be described as (infinite) constraint systems over complete lattices. Such constraint systems can be handed off to local solvers, that try to apply demand-driven computation to find (partial) solutions. In this paper, we took on-demand computation a step further: we have proposed and investigated a mechanism for making interprocedural static analysis more demand-driven by distributing map domains between different constraint system variables. As the first example, we successfully apply our approach to the analysis that consists of a series of helper analyses, where some are only partially needed.

For the rest of the paper, we focused on the points-to analysis where the analysis domain is a mapping from program variables. We conclude that the mechanism is suitable when using the call string approach. The functional approach is, however, not suitable, as it inherently adds dependencies to other keys of the map-domain—defeating our efforts.

We have considered the single-threaded case and, with the help of side-effecting constraint systems, also the multithreaded case. The experimental evaluation shows that the presented mechanism works in general. It is, however, a practical challenge to write the transfer function in such a way to keep the overhead of distributing map keys to different constraint system variables small. As future work, we will explore potential pre-processing steps for the constraint system generation to reduce the overhead incurred. Instead of using functions as the right-hand sides of the constraint system, we will use a modification of strategy trees [21], which can be inspected and optimized.

Acknowledgement. This work was supported by the Estonian Research Council grant PUT PSG61.

References

References


