Abstract
When writing a program, the programmers may wish to verify certain properties of this program. For example, will my variable X always be an integer? To answer these questions we use program analysis. Constraint-based program analysis is a form of static program analysis. This means that all of these questions about the program’s run-time behavior are answered at compile time. This paper seeks to provide a survey of constraint-based program analysis including: a description of the approach, research done on the topic, applications of the topic, and future directions.

1. Introduction
The idea for constraint-based program analysis was first introduced by John C. Reynolds in 1969 in his paper Automatic Computation of Data Set Definitions [12]. At the time, systems which provide fast and flexible data representations require the programmer to give the range of variables, parameters, and functions by detailed data structure definitions. Reynolds’s insight was that this information is redundant, as much of it can be inferred. His paper discusses a method for giving data set description of the output and input of a function given the program of the function, this was done specifically in the LISP programming language. This idea has been expanded into the modern program analysis approach of constraint-based program analysis. The rest of the paper will primarily focus on the most common form, set constraint-based program analysis, but others will be mentioned.

One of the main draws of constraint-based program analysis is the ability of the constraints to separate specification from implementation [1]. There are two basic steps for performing analysis: constraint generation and constraint resolution. As its name seems to imply, constraint generation is the processes of creating the constraints from the to-be-analyzed program. The resolution is the implementation of these constraints. This is desirable for two reasons. First, the soundness of the constraint system depends only on system used, so the resolution algorithm bears no impact. Additionally, the algorithms used to perform constraint solving may be written independent of the eventual constraint system used. This allows for toolkits to be produced for solving constraint systems (examples of these are discussed later in the paper).

Another key feature of constraint-based program analysis is that constraints tend to flow naturally from the source code. Each segment of the program may contribute its own constraints. It is then the interplay of the entire system of constraints which begets the eventual solution. The isolation of the code segments makes the constraint generation and interpretation a conceptually simple task[1]. A final draw is that constraints allow for very complex implementation. The set constraints language is rich enough from all commonly used data types. Also, set constraint-based program analysis can be used to perform other forms of program analysis such as dataflow analysis and type inference. First, I will discuss the basics of constraint generation and resolution.

Basically, set constraints define relationships between sets of terms. The form of a set constraint is syntactically identical to what would be expected of sets. For example, a constraint could be of the form \( X \subseteq Y \) where \( X \) and \( Y \) are set expressions. For the constraints we introduce a set of constructors \( C \) and a set of set-valued variables \( V \). For each \( c \in C \) we say that \( a(c) \) is the arity of \( c \), that is the number of arguments the constructor takes. Thus it follows that if \( a(c) = 0 \) then \( c \) is a constant. The expressions are defined by the following grammar:

\[
E ::= \alpha [0] | E_1 \cup E_2 | \neg E_1 | c(E_1, ..., E_{a(c)}) | \epsilon^{-1}(E_1)
\]

Here, \( \alpha \) is a variable, that is \( \alpha \in V \), and \( c \) is a constructor, that is \( c \in C \). We can say now that set expressions are sets of terms (for example the set of values a variable may hold). A term is represented as \( c(t_1, ..., t_{a(c)}) \). As before, \( c \in C \), and now \( t_i \) is a term. The base case of a term is a constant. We now have that the set of all terms is the Herbrand Universe \( H \). I now introduce assignments. An assignment is a mapping \( V \rightarrow 2^H \) that assigns sets of terms to variables, an assignment is represented by \( \sigma \). This is to be interpreted similar...
to a Java object being assigned to a variable. We derive the meaning of set expressions by extending assignments from variables to set expressions in the following manner[1]:

\[\sigma(0) = \emptyset\]

\[\sigma(E_1 \cup E_2) = \sigma(E_1) \cup \sigma(E_2)\]

\[\sigma(E_1 \cap E_2) = \sigma(E_1) \cap \sigma(E_2)\]

\[\sigma(-E_1) = \text{H} - \sigma(E_1)\]

\[\sigma(c(E_1, ..., E_n)) = \{c(t_1, ..., t_n) | t_i \in \sigma(E_i)\}\]

\[\sigma(c^{-1}(E)) = \{t_i | \exists c(t_1, ..., t_n) \in \sigma(E), 1 \leq i \leq n\}\]

Aiken in [1] presents examples of the expressive power of this language. I will also provide them here.

Consider the constraint \(\beta = \text{cons}(\alpha, \beta) \cup \text{nil}\) where for set constraints \(X, Y, X \cap Y \subseteq X \cap Y \subseteq Y\). We let \text{cons} and \text{nil} have their normal interpretations. The solution for \(\beta\) is the set of all lists with elements in \(\alpha\). Therefore, it is very easy to represent a list data structure using set constraints. As a more complex example we now present red-black trees. Recall that a red-black tree is a binary search tree with the following properties:

1. Every node is either red or black
2. Every leaf is black
3. Every red node has 2 black children
4. Every path from the root to a leaf has the same number of black nodes

Unfortunately, set constraints cannot be used to satisfy property 4, but 1-3 can be represented in the following way. Here, we let \(\alpha\) represent a subtree with a black root, and \(\beta\) as a subtree with a red root. We let \text{red}, \text{black} \in C be binary constructors.

\[\alpha = \text{black}(\alpha \cup \beta, \alpha \cup \beta) \cup \text{blackleaf}\]

\[\beta = \text{red}(\alpha, \alpha)\]

Now that we have discussed the generation of constraints, now we will discuss the resolution of constraints. A system of constraints is defined as a conjunction of constraints \(\bigwedge_i X_i \subseteq Y_i\) where \(X_i, Y_i\) are set expressions. A solution is an assignment \(\sigma\) such that \(\bigwedge_i \sigma(X_i) \subseteq \sigma(Y_i) \equiv \text{true}\). All algorithms to find solutions work in this basic way: given an initial set of constraints it is simplified using pre-defined transformation rules until the system is in solved form[3]. These rules, however, must satisfy certain properties[9]:

1. Each transformation step is sound, meaning that after a transformation is applied to a conjunction, the solution to resulting conjunction is the same as that of the initial conjunction.
2. When transformations are exhaustively applied (continue applying until none are applicable), the algorithm is guaranteed to either terminate or detect an inconsistency.
3. The the set of transformations is complete, meaning that once none apply all desired information about the solutions of the constraints has been obtained.

Solving these can be done in various ways (as stated before, constraint resolution is independent of the source program or information to be abstracted, giving much freedom). Perhaps the most common method for doing this is done with constraint graphs, which is the method this paper will primarily discuss, to be discussed in the following section.

The remainder of the paper is organized as follows: section 2 contains a discussion of the algorithms used for constraint resolution, with section 2.1 applying heuristics to reduce computation time in practice and 2.2 addressing space complexity concerns, section 3 will address two ways in which constraint-based program analysis has been applied to solve real-world problems, section 4 describes the toolkits for implementing constraint-based program analysis in real-world systems, and section 5 contains a conclusion with a short discussion of future directions.

2. Algorithms and complexity for Constraint Resolution

As mentioned before, to solve a system of constraints one exhaustively applies a given set of substitution rules. One algorithm for performing this was presented by Aiken and Wimmers in [5]. This is done on the language

\[E ::= 0 | 1 | \alpha | c(E_1, ..., E_{\alpha(e)}) | E_1 \cup E_2 | E_1 \cap E_2 | \neg E_1\]

With 0 being the empty set and 1 being the set of all terms. Space does not permit a discussion of the details of this approach, however at a conceptual level the algorithm operates as follows:

1. Reduce the input system to a one-level system. A one-level system being one in which all constraints are of the form \(X \subseteq 0\) and \(X\) has no unions, no nested constructors, and all negations are on variables.
2. Reduce a one-level system to a cascading system. A cascading system insures that \(c(X, Y) \subseteq 0 \Rightarrow X \subseteq 0\) or \(Y \subseteq 0\), as well as transitivity.
3. Reduce the cascading system of constraints to a system of equations.
4. Reduce the system of equations into solved form.

To perform this, we may view the system of constraints as a directed graph \(G(C)\), where \(C\) is the system of constraints. The nodes of the graph are set expressions, the edges are atomic constraints. A constraint is atomic if one side is a set variable (either the left or right side). An application of a resolution rule is represented by appropriately adding new edges to the graph. In general, a system of constraints \(C\) with \(v\) variables can be reduced to solved form in \(O(v^2 |C|)\)[3].

One particular way to represent these graphs is in inductive form (as presented in [3]), which gives the advantage of
reduced transitive edges which in turn serves to reduce run time and space complexity. In this form there are two types of edges: successor edges and predecessor edges. If we are given the constraint \( X \subseteq Y \) then the graph may either have the successor edge \( Y \in \text{succ}(X) \) or the predecessor edge \( X \in \text{pred}(Y) \). The decision for which to use is based on a fixed ordering of the variable \( o \). This ordering is generated randomly. So say we have that \( o(X) > o(Y) \), so \( X \) comes after \( Y \) in \( o \), then we have the constraint \( X \subseteq Y \) as a successor edge on \( X \), otherwise if \( X \) comes first the constraint is represented as a predecessor edge on \( Y \). It then follows that the transitive closure for a graph in inductive form is \( L \in \text{pred}(X) \land R \in \text{succ}(X) \Rightarrow L \subseteq R \). This, along with the following resolution rules, produce a solved graph.

\[
C \cup \{ X \subseteq X \} \Leftrightarrow C
\]

\[
C \cup \{ c(E_1, ..., E_{a(c)}) \subseteq c(E_1', ..., E'_{a(c)}) \} \Leftrightarrow C \cup \bigcup_i \{ E_i \subseteq E_i' \}
\]

\[
C \cup \{ c(E_1, ..., E_{a(c)}) \subseteq \text{proj}(c, i, E) \} \Leftrightarrow C \cup \{ E_i \subseteq E \}
\]

\[
C \cup \{ d(E_1, ..., E_{a(d)}) \subseteq \text{proj}(c, i, E) \} \Leftrightarrow C \text{ if } c \neq d
\]

\[
C \cup \{ c(...) \subseteq d(...) \} \Leftrightarrow \text{no solution if } c \neq d
\]

Here, we define the project \( \text{proj}(c, i, E) \) as a construction taking the \( i^{th} \) component of \( c \) (call this \( E_i \)) and adding the constraint \( E_i \subseteq E \).

In general, solving set constraints is computationally difficult, having been shown to be \( \text{NEXPTIME-complete} \) [4]. In order to improve this complexity one must either use heuristics or present restrictions in order to work on more specific problems (as was done in [3]). Many have been found with tractable solutions, but it is still an open problem to find more[9].

### 2.1 Optimizations

Since the time complexity of solving constraints is quite high, it is necessary to find optimization heuristics in order to reduce computation time. Approaches such as periodic simplification of the constraint system have been shown to help, but do not do enough to make the approach scalable. This is primarily done based on intuition about the structure of constraint graphs in practice.

In [7] the authors hypothesize that cycle elimination in constraint graphs will provide substantial reduction in constraint resolution time. The constraint graphs used here are inclusion constraint graphs. That is, constraints such as \( X \subseteq Y \subseteq Z \) are represented in the graph as nodes \( X, Y, Z \) and directed edges \( (X, Y), (Y, Z) \). It is resolved by adding new edges to express the implicit edge relationships in the system, for example the transitive edge \( (X, Z) \) for the implied constraint \( X \subseteq Z \). They specifically use the inductive form of graphs as presented in the preceding section as well as in [3]. The reason being that this structure more drastically improves with their approach (though they have also shown that the standard from also improves). A cyclic constraint is defined as a set of constraints \( X \subseteq Y \subseteq Z \subseteq \ldots \subseteq X \). The result of such a set of constraints is that all the variables must have the same set of solutions, so the listing of all such constraints is unnecessary. The cycle within the graph can be eliminated by collapsing all of the variables into a single variable.

There are multiple approaches for achieving cycle elimination. One could check for cycles with every added edge in the graph. This, however, is incredibly expensive, as the algorithm must do a depth-first search for each update. The solution presented is to do partial cycle elimination, meaning that only some paths are traversed while performing cycle detection. Recall the construction of the constraint graph as presented above. We have the relationship:

\[
X \subseteq Y = \begin{cases} 
X \rightarrow Y & \text{if } o(X) > o(Y), \\
X \leftarrow Y & \text{if } o(X) < o(Y), \\
\end{cases}
\]

So now we have a cycle represented as:

\[
X_1 \longrightarrow X_2 \longrightarrow X_3
\]

The algorithm for partial online cycle detections works as follows. When a successor edge \( X \rightarrow Y \) is added, search through all predecessor edges starting from \( X \) for a predecessor chain \( Y \rightarrow X \). If a predecessor edge \( X \rightarrow Y \) is added, search along all successor edges starting from \( Y \) for a successor chain \( X \rightarrow Y \). A predecessor chain being a simple patch \( (X_0, ..., X_k) \) consisting entirely of predecessor edges. A successor chain is similar.

In practice this solution detected 80% of all variables involved in cycles, and provided an overall speedup over the standard approach by a factor of 50 in large programs.

Another approach for an optimization is projection merging, presented in [8]. This problem is again addressed in inductive form. The intuition is that when constraint edges are added, we sometimes get redundencies. The insight for addressing this problem follows from the utilization of inductive form graphs: that they are more likely to add edges between constructed terms rather than edges between two variables. While space does not permit discussion of the specifics, here is an overview of the approach. The first step is to extend the set of closure rules to include new rules \textbf{Proj-Creation} and \textbf{Proj-Transitive}. They then choose an ordering of variables which ensures that the application of the modified rules will finish quickly.

### 2.2 Space Complexity

In addition to high time complexity, the space complexity of constraint-based program analysis can hinder its scalability.
Here I will briefly mention a few approaches for reducing this complexity [9].

1. **Polymorphism**: When performing constraint resolution with polymorphic types it may be necessary to do repeat analysis in different contexts. This can be solved by constraint simplification by substituting the polymorphic constrained type with an equivalent polymorphic type of fewer variables.

2. **Simplification**: Often a constraint system contains extraneous information. Simplification seeks to remove this. One such way of doing this is to collapse cycles, as discussed in the previous section.

3. **Separation**: One can analyze subtrees of an abstract Syntax tree independently (note that constraint systems are a generalization of abstract syntax trees). This allows for each to be simplified independently, thus making the merge of them more simple.

4. **Sparse Constraint representation**: Much of the space complexity comes from the representation of transitive constraints. One approach is to not remember these, but rather recompute them as needed. This obviously has a time complexity tradeoff.

### 3. Applications

One application of constraint-based program analysis is soft-typing [6]. Dynamically typed languages such as Lisp and Scheme make it notoriously difficult to perform static type checking, sometimes forcing all type-checking to be done at run-time. The authors in this paper present a method for performing type inference on the dynamically typed languages, called soft-typing. The result is an approach which is powerful and simple, claiming to "infer the most accurate types of values."

The novelty of this approach lies in the conditional types. Consider the expression

\[
\text{case } e_1 \text{ of true } : e_2, \text{ false } : e_3
\]

The conditional types can express that \(e_2\) is evaluated only when \(e_1\) is true and \(e_2\) is evaluated only when \(e_1\) is false. So essentially, the conditional types perform control-flow analysis via type checking. Ultimately, the approach is to implement soft typing by solving a system of type inclusion constraints over a type language including function types, a least type \(\tau_0\), a greatest type \(\tau_1\), constructor types, union, intersection, recursive types, and conditional types. This language is given as:

\[
\begin{align*}
\tau &::= \tau_1 \rightarrow \tau_2 | e(\tau_1, ..., \tau_{a(c)}) | \alpha | \tau_1 \cup \tau_2 | \tau_1 \cap \tau_2 | \tau_1 ? \tau_2 | 0 | 1 \\
\sigma &::= \tau | \forall \alpha_1, ..., \alpha_n, \tau | where \{ ..., \tau_1 \subseteq \tau_2, ... \}
\end{align*}
\]

With \(\tau\) as an unquantified type and \(\sigma\) as a quantified type. All notation is the same as in previous sections, with the additions of function types \((\tau_1 \rightarrow \tau_2\), meaning the set of functions mapping elements of type \(\tau_1\) to those of type \(\tau_2\)), and conditional type \((\tau_1 ? \tau_2\), reading "\(\tau_1\) if \(\tau_2\)").

This approach has been implemented and tested in the language FL. It is worth noting that the system was only usable when optimizations were applied, even when only considering small programs.

Another application of Constraint-based program analysis, as discussed in [10], is compile time analysis of ML programs. In this study the author sought to perform program analysis on a functional programming language. The motivation being that the approach discussed would be used in the future to perform compiler optimizations. The author identifies the follow as important compilation issues for functional programming languages: array bounds checks, control flow, redundant tests, inlining and specialization. The author also points out that most research in program analysis only targets a specific optimization, so to address multiple optimizations one must apply multiple analyses. Therefore the author uses set constraint-based program analysis in order to use a single analysis in order to answer multiple needs. Specifically, this analysis treats all program variables as sets of values. In the end, the author produces a program analysis that can be computed in \(O(n^3)\) time and has the following properties, all indicative of previously-discussed advantages of program analysis:

1. The analysis provides accurate information about the program.
2. The set constraints have a simple definition, so the analyses are intuitive.
3. It is flexible in that new functionality can be easily added.

It is also worth noting that this application employs many of the results discussed previously in this paper: constraint generation, constraint resolution, polymorphism in order to reduce space complexity, and soft typing. Further applications from this result have been achieved in partial evaluation and array bounds checking.

### 4. BANE and BANSHEE

As mentioned in Section 1, one key advantage of constraint-based program analysis is the separation of constraint generation from constraint resolution, allowing each to be performed out of context from the other. This is fully utilized in the toolkits Berkley ANalysis Engine (BANE) [2] and its successor Banshee [11]. Being as BANE came first, I will discuss BANE first.

The goal of BANE was to lower the overhead burden of running program analyses by providing a simple toolkit for running them. In order to implement BANE the user must only write code to generate the desired constraints from the to-be analyzed program and then, after running BANE, interpret the solutions. Essentially, BANE takes care of the entirety of the constraint resolution step. In implementing the
In order to achieve impressive results, open areas are similar to those seen in [10] and [6], where the plying constraint-based program analysis is feasible. Other is much research in finding restraints on systems so that ap-

problems are characterized by a language of expressions, a constraint relation, a solution space, and an implementation strategy. A key advantage to this is allowing for mixed constraints, which means using constraints of multiple sorts in a single application. BANE also successfully made its program analysis scalable to large systems. This was achieved by implementing many of the optimizations discussed previously, including separation as mentioned in section 2.2.

Banshee was meant as the improved successor to BANE. Banshee was written from the ground up, but it inherits many features from BANE, notably its support for mixed constraints. The main motivation for Banshee was that BANE could not perform as quickly as hand-written constraint resolution algorithms, and small changes to code would require the entirety of the program analysis to be repeated. These issues are resolved with the following new features:

• Capability for the creation of specialized resolution engines: this is done by the analysis designer providing constructor signatures.

• Backtracking: this allows for incremental analysis, so when small changes are made only parts of the program are re-analyzed.

• Serialization and Deserialization of constraint systems: allows for the ability to save and load constraint systems.

The authors claim that Banshee has reached the status of being capable of producing production-quality program analyses. This is evidenced by its application on partial evaluation of graphics programs, as a part of a software updateability analysis tool, a type inference system for Prolog, and for pointer analysis in the gcc compiler.

5. Conclusion

Currently, many open problems deal with the time and space complexity of the approaches. While general cases of constraint-based program analysis are very difficult (as mentioned, general satisfiability of constraints is NEXPTIME-complete), there are many subclasses of problems with tractable solutions (see the system in [10]). Therefore, there is much research in finding restraints on systems so that applying constraint-based program analysis is feasible. Other open areas are similar to those seen in [10] and [6], where the research is simply finding new ways to generate constraints in order to achieve impressive results.

Hopefully this paper has expanded the reader’s understanding of constraint-based program analysis. This technique is one which is fairly intuitive and quite applicable in practice. The summary of the technique is to (1) create constraints which represent the desired information generated from the desired program text and then (2) resolve and simplify these constraints in order to make inferences on the program. The main strength of this technique is that (1) and (2) may be discussed separately. All of the generalizability and significant results presented in this paper are made possible by this fact. For example, the optimizations provided would not be applicable as generally as they are if constraint resolution were more application-specific, constraint-based analysis could not be used for soft-typing if there were not flexibility in constraint generation, and there could be no general toolkits for performing this unless both were separate from one another.

For the purposes of the project, the following papers were given a thorough read: [1], [3], [5], [6], [7], and [9]. The remainder of the papers were given a more superficial read.

References


