Introduction to Set Constraint-Based Program Analysis

Alexander Aiken*
EECS Department
University of California, Berkeley
Berkeley, CA 94702-1776
aiken@cs.berkeley.edu

1 Introduction

Program analysis is concerned with automatically extracting information from programs. Program analysis is a large topic, with a long history and many applications, particularly in optimizing compilers and software engineering tools. As might be expected of any broad area, there are a number of distinct approaches to program analysis.

This paper provides an overview of *constraint-based* program analysis. While much has been written about constraint-based program analysis in recent years, there is relatively little material to assist outsiders who wish to learn something about the field. Two survey papers cover the computational complexity of various constraint problems that arise in program analysis [Aik94, PP97]. The purpose of the present work is to motivate the use of constraints for program analysis from the perspective of the applications of the theory.

Program analysis using constraints is divisible into *constraint generation* and *constraint resolution.* Constraint generation produces constraints from a program text that give a declarative specification of the desired information about the program. Constraint resolution (i.e., solving the constraints) then computes this desired information. In the author's view, the constraint-based analysis paradigm is appealing for three primary reasons:

- *Constraints separate specification from implementation.* Constraint generation is the specification of the analysis; constraint resolution is the implementation. This division helps to organize and simplify understanding of program analyses. The soundness of an analysis can be proven solely on the basis of the constraint systems used—there is no need to resort to reasoning about a particular algorithm for solving the constraints. On the other hand, algorithms for solving classes of constraint problems can be presented and analyzed independent of any particular program analysis. General results on solving constraint problems provide "off-the-shelf" tools for program analysis designers.

- *Constraints yield natural specifications.* Constraints are (usually) local; that is, each piece of program syntax contributes its own constraints in isolation from the rest of the program. The conjunction of all local constraints captures global properties of the program being analyzed.

*This work was supported by NSF National Young Investigator award CCR-9457812. This version includes corrections suggested by Manuel Fähndrich.

- *Constraints enable sophisticated implementations.* The constraint problems that arise in program analysis have a rich theory that can be exploited in implementations. We shall only touch on this subject in this paper.

We first briefly discuss the long history of the use of constraints in program analysis, which predates the current interest in the area by many years (Section 2). The overview proper begins with the introduction of *set constraints*, a widely used constraint formalism in program analysis and the one with which the author is best acquainted (Section 3).

The balance of the paper shows that three classical problems—standard dataflow equations, simple type inference, and monomorphic closure analysis—can be viewed as instances of set constraint problems (Section 4). Each of these three very basic analyses have been developed by different communities of people over extended periods of time, and to our knowledge no formal connection between the problems has been noted previously in the literature. Our main aim in choosing these problems, however, is that we assume most readers are familiar with at least one of them and thereby are afforded an easy path to appreciation of the constraint-based analysis perspective. We also present one simple variation of type inference suggestive of the expressive power provided by set constraints (see Section 4.3).

To give some insight into the algorithmic issues involved in a general constraint-based analysis system we give constraint resolution algorithms for the constraint systems arising from the three example analyses. It is important to realize that in different applications we are interested in different notions of constraint solvability. Depending on the application, we may be interested in only knowing a particular solution (e.g., the least solution) or in calculating all solutions.

Set constraints provide one of the most general decidable theories known for constraint-based program analysis, and the essential issues of constraint-based analysis can be illustrated easily using set constraints. However, we do not wish to give the impression that set constraints are the only useful constraint theory for program analysis. In addition, there are of course other approaches to program analysis not based on constraints. Other constraint formalisms, altogether different approaches, as well as the place of constraint-based program analysis in the general theory of *abstract interpretation*, are discussed in Section 6.

2 History

Using constraints in program analysis is not a new idea. The earliest example we are aware of is due to Reynolds, who proposed an analysis of Lisp programs based on the resolution of inclusion constraints in 1969 [Rey69]. Similar ideas (but based on grammars rather than constraints) were developed independently later by Jones and Muchnick [JM79]. Dataflow equations and type equations, two examples that we shall investigate in greater depth in Section 4, also have a long history. Dataflow equations form the basis of most classical algorithms for flow analysis used in compilers for procedural languages (most notably C and FORTRAN). Type equations are the basis of type inference for functional languages and for template-style polymorphism in object-oriented languages.

While the idea of program analysis using constraints is not new, there has been a dramatic shift in the research perspective in recent years. Formerly, each of the problem areas described above was viewed as a separate line of research, with its own techniques, problems, and terminology. Efforts to hybridize or extend these techniques met with considerable difficulty, at least in part because it was unknown whether the resulting constraint problems could be solved. Today it is understood that these problems are related, and that much can be gained by viewing the problems as instances of a more general setting. In fact, techniques from each of the classical algorithms may be combined quite freely to create new program analyses.

To make the advantages of the constraint perspective concrete, we use another classical problem for illustration. Most compilers perform *register allocation* to assign machine registers to program variables. Consider the following fragment of imperative code, where program variables are named `a,b,c,` and so forth:

```
a := c + d
e := a + b
f := e - 1
print(f)
```

A *valid register assignment* is a mapping from variable names to register names that preserves program semantics. If the register names are `r1, r2, r3, ...`, then the program under one valid register assignment may be:

```
r1 := r2 + r3
r4 := r1 + r5
r1 := r4 - 1
print(r1)
```

The difficulty in register allocation is that there are usually more program variables than there are registers to hold them. In the example above, six variables are mapped into five registers, with variables `a` and `f` sharing register `r1`. In general, a valid register allocation may not even exist for a given program. In this case, the number of variables in the program can be reduced by *spilling* some variables by inserting code to save and restore these variables to and from main memory.

The register allocation problem was already recognized in the FORTRAN I compiler in the 1950's, but the solution techniques were *ad hoc* and not entirely effective. By the 1970's it was realized that the weakness of contemporary register allocation was a limiting factor in the development of optimizing compilers. A breakthrough came in the late 1970's when Chaitin proposed a register allocation heuristic based on graph coloring [CAC$^+$81]. The significance of the contribution can be judged by the fact that this technique was the subject of one of the first software patents. Chaitin's insight was to formulate register allocation as a constraint problem.

A variable `x` is said to be *live* at a program point p if `x` is referred to at some program point later in the execution ordering than p with no intervening assignment to `x`. Otherwise `x` is said to be *dead.* Consider an assignment statement `y` := .... A basic observation about register allocation is

> *If variable* `x` *is live when variable* `y` *is assigned, then* `x` *and* `y` *cannot be held in the same register.*

In the example above, we have implicitly assumed that `a` is dead at the point where `f` is assigned, allowing reuse of `a`'s register to hold the value of `f`.

This observation suggests the following natural constraint problem. Let *Reg* : *Variables* → *Registers* be a register assignment. The constraints on *Reg* are

$$\mathit{Reg}(\mathtt{x}) \neq \mathit{Reg}(\mathtt{y}) \Leftrightarrow \mathtt{x} \text{ is live where } \mathtt{y} \text{ is assigned}$$

This formulation neatly captures the constraints under which a register assignment is valid. The next problem is to compute register assignments. The constraints naturally specify a graph with one node for each variable and an edge $(\mathtt{x}, \mathtt{y})$ for each inequality constraint $Reg(\mathtt{x}) \neq Reg(\mathtt{y})$. A graph is *k-colorable* if each node of the graph can be assigned a color different from the color of all of its neighbors in such a way that no more than k colors are used. Finding a register assignment with k registers is equivalent to finding a k coloring of the constraint graph.

By the time of Chaitin's work, it was already known that graph coloring is an NP-complete problem, and therefore that efficient exact solutions were very unlikely to be found. Chaitin proposed a simple heuristic for coloring the graph based on another observation:

> *If a node* `x` *has fewer than k incident edges, then the graph is k-colorable if and only if the graph obtained by removing* `x` *and its edges is k-colorable.*

That is, if `x` has fewer than k neighbors, then there is always a color for `x`, no matter how the rest of the graph is colored. In cases where the heuristic fails to color the entire graph (i.e., a point is reached where all nodes have k or more neighbors) it is necessary to choose a variable to spill. While subsequent work extends the heuristics for coloring and spilling, graph coloring remains the best framework known for register allocation after nearly 20 years.

This rather old example illustrates all of the advantages of using constraint formulations in program analysis. The constraint formulation as inequalities separates the specification of the problem from its implementation, and most importantly gives a global characterization of the conditions to be satisfied. The abstract constraint problem, now free of the details of the particular program and programming language, can then be addressed by appropriate techniques, in this case graph coloring. Note that the constraint resolution algorithm proceeds in a manner that has no direct relationship to program structure, and that if one were to actually view the sequence of allocation decisions made by the greedy coloring heuristic it would jump around from point to point in the program with no apparent pattern. If we were to attempt formulating directly an algorithm that was defined, e.g., by induction on the program syntax, it is unlikely we would arrive at something as effective as converting the problem to a constraint representation.

The reader may find register allocation heuristics a peculiar choice for a historical example of program analysis. After all, graph coloring register allocation is not usually even regarded as a program analysis problem, let alone a constraint-based one. However, it is clear that the constraint formulation was central in developing the technique. Register allocation is interesting for another reason. To our knowledge, it is the only significant application of *negative* constraints (i.e., inequalities) to program analysis in the literature.

3 Set Constraints

This section gives a brief overview of set constraints and the state of knowledge on set constraint problems. In Section 4 we illustrate connections between disparate program analysis problems using the language of set constraints.

Set constraints describe relationships between sets of terms. A set constraint has the form $X \subseteq Y$, where X and Y are *set expressions*. Let C be a set of constructors and let V be a set of set-valued variables. Each $c \in C$ has a fixed arity $a(c)$; if $a(c) = 0$ then c is a constant. The set expressions are defined by the following grammar:

$$E ::= \alpha \,|\, 0 \,|\, E_1 \cup E_2 \,|\, E_1 \cap E_2 \,|\, \neg E_1 \,|\, c(E_1, \ldots, E_{a(c)}) \,|\, c^{-i}(E_1)$$

In this grammar, α is a variable (i.e., $\alpha \in V$) and c is a constructor (i.e., $c \in C$). In the standard interpretation, set expressions denote sets of *terms*. A term is $c(t_1, \ldots, t_{a(c)})$ where $c \in C$ and every t_i is a term (the base cases of this definition are the constants). The set of all terms is the Herbrand universe H. An *assignment* σ is a mapping $V \to 2^H$ that assigns sets of terms to variables. The meaning of set expressions is given by extending assignments from variables to set expressions as follows:

$$\begin{aligned}
\sigma(0) &= \emptyset \\
\sigma(E_1 \cup E_2) &= \sigma(E_1) \cup \sigma(E_2) \\
\sigma(E_1 \cap E_2) &= \sigma(E_1) \cap \sigma(E_2) \\
\sigma(\neg E_1) &= H - \sigma(E_1) \\
\sigma(c(E_1, \ldots, E_n)) &= \{c(t_1, \ldots, t_n) | t_i \in \sigma(E_i)\} \\
\sigma(c^{-i}(E)) &= \{t_i | \exists c(t_1, \ldots, t_n) \in \sigma(E), 1 \leq i \leq n\}
\end{aligned}$$

A *system of set constraints* is a finite conjunction of constraints $\bigwedge_i X_i \subseteq Y_i$ where each of the X_i and Y_i is a set expression. A *solution* of a system of set constraints is an assignment σ such that $\bigwedge_i \sigma(X_i) \subseteq \sigma(Y_i)$ is true. A system of set constraints is *satisfiable* if it has at least one solution.

The term "set constraints" was coined by Heintze and Jaffar [HJ90], who were the first to recognize and formalize set constraints in their full generality. It is a remarkable fact about many set constraint problems that not only is it decidable whether or not a system of constraints has a solution, but that all (potentially infinitely many) solutions can be given a finite representation. In their original paper, Heintze and Jaffar showed that a restricted class of set constraints could be solved and the solutions finitely presented.[1]

A natural and interesting subclass of set constraints excludes projections but includes all other operations. An algorithm that exhibits all solutions of such constraints first appears in [AW92]. Subsequently, many alternative proofs of this result and connections to other disciplines were discovered, including tree automata [GTT92] and graph theory [AKVW93]. A particularly elegant result shows that set constraints without projections are equivalent to the monadic class of predicate logic [BGW93].

Including unrestricted projections in a complete theory turns out to be a difficult problem. A series of papers by a variety of authors show increasingly powerful systems of constraints to be decidable [GTT93, BGW93, CP94a, AKW95]. Charatonik and Pacholski finally show that the full set constraint language is decidable in [CP94b].

Showing decidability is, of course, a necessary first step in obtaining practical algorithms. Beyond decidability, we would like efficient algorithms and algorithms that compute finite representations of solutions. In these areas the state of knowledge is incomplete. Currently, the algorithms that compute finite representations of the solutions of set constraints cannot handle unrestricted projections. Furthermore, the complexity of solving general set constraints is high. Satisfiability of set constraints is NEXPTIME-complete; in fact, it remains NEXPTIME-complete even if projections are eliminated.

The complexity results strongly suggest that analyses based on solving set constraints in their full generality are infeasible. However, there are many very useful polynomial time fragments of the full theory, and it is these tractable sub-theories that are our focus in this paper.

3.1 Expressive Power

From the definition above, it is easy to see that the set expressions consist only of elementary set operations plus constructors—simply put, it is a set theory of terms. The constraint language is rich enough,

[1]It is also worth noting that for some variations of set constraints, in particular with the addition of function spaces, no complete resolution algorithm is known for the general case.

however, to describe all of the data types commonly used in programming, and this is the property that makes set constraints a useful tool for program analysis. For example, programming language data type facilities provide "sums of products" data types, which means simply unions of (usually distinct) data type constructors. All such data types can be expressed as set constraints.

Let $X = Y$ stand for the pair of constraints $X \subseteq Y$ and $Y \subseteq X$. Consider the constraint

$$\beta = \texttt{cons}(\alpha, \beta) \cup \texttt{nil}$$

If `cons` and `nil` are interpreted in the usual way, then the solution of this constraint assigns to β the set of all lists with elements drawn from α. This example also shows that a special operation for recursion is not required in the set expression language—recursion is obtained naturally through recursive constraints.

We have not said whether we mean our lists above to be strict (as in most languages) or non-strict (as in lazy functional languages). Set constraints can be used for either, although different models are required for strict and non-strict constructors. In this paper we wish to avoid most of the complexities of discussing models, so we simply observe that for a non-strict `cons` the following identity holds:

$$\texttt{cons}(X, Y) \subseteq \texttt{cons}(X', Y') \Leftrightarrow X \subseteq X' \wedge Y \subseteq Y'$$

For a strict `cons` one must naturally account for strictness, namely that $\texttt{cons}(0, Y) = 0$ for all Y (and similarly for a 0 in the second position). Thus the identity for a strict `cons` is more complex:

$$\texttt{cons}(X, Y) \subseteq \texttt{cons}(X', Y') \Leftrightarrow (X \subseteq X' \wedge Y \subseteq Y') \vee X = 0 \vee Y = 0$$

It is by applying equivalences such as these that set constraint solvers solve set constraints (see Section 5). By choosing the appropriate resolution rules either strict or non-strict constructors can be modeled faithfully; in fact, it is possible to distinguish individual arguments of constructors as strict or non-strict, though we know of few applications for such generality. Because of the disjunction on the right-hand side of the $\Leftrightarrow$, it is in general more expensive to resolve constraints involving strict constructors than constraints using only non-strict constructors.

The set of non-nil lists (with elements drawn from α) can be defined as $\gamma = \beta \cap \neg\texttt{nil}$, where β is defined as above. The set γ is useful because it describes the proper domain of the function that selects the first element of a list; such a function is undefined for empty lists. This example also illustrates that set constraints can describe proper subsets of standard sums of products data types.

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 two black children.

4. Every path from the root to a leaf has the same number of black nodes.

Together these properties imply that a red-black tree of n nodes has height at most $2\log(n + 1)$, so red-black trees are well-balanced trees. Set constraints can describe properties (1)-(3) of red-black trees. In the following equations, the set α describes subtrees rooted at black nodes and β describes subtrees rooted at red nodes. `Red` and `black` are both binary constructors:

$$\begin{aligned} \alpha &= \texttt{black}(\alpha \cup \beta, \alpha \cup \beta) \cup \texttt{blackleaf} \\ \beta &= \texttt{red}(\alpha, \alpha) \end{aligned}$$

Property (4) of red-black trees cannot be described by set constraints. This follows from the fact that the solutions of set constraints are always describable by regular equations(see Section 5).

The final, admittedly contrived, example shows a non-trivial system of constraints where some work is required to derive the solutions. Consider the universe of the natural numbers with one unary constructor `succ` and one nullary constructor `zero`. Let the system of constraints be:

$$\texttt{succ}(\alpha) \subseteq \neg\alpha \bigwedge \texttt{succ}(\neg\alpha) \subseteq \alpha$$

These constraints say that if $x \in \alpha$ (resp. $x \in \neg\alpha$) then $\texttt{succ}(x) \in \neg\alpha$ (resp. $\texttt{succ}(x) \in \alpha$). In other words, these constraints have two solutions, one where α is the set of even natural numbers and one where α is the set of odd natural numbers. The solutions are described by the following equations:

$$\alpha = \texttt{zero} \cup \texttt{succ}(\texttt{succ}(\alpha))$$
$$\alpha = \texttt{succ}(\texttt{zero}) \cup \texttt{succ}(\texttt{succ}(\alpha))$$

The two solutions are incomparable; in general, there is no least solution of a system of set constraints.

3.2 Extensions

There are extensions of set constraints that have proven useful in various applications. The most important extensions are surveyed here.

3.2.1 Function Space

Function spaces $X \to Y$ can be added to the set expressions. In an appropriate model, the meaning of $X \to Y$ is

$$X \to Y = \{f | x \in X \Rightarrow f(x) \in Y\}$$

Note that semantically $\to$ is not a labelled cross product of the domain and the range; thus the term semantics of set expressions given above are not adequate to model function spaces. A suitable domain can be constructed using standard techniques of denotational semantics and, given such a domain, set constraint resolution techniques still apply, although so far as is known additional restrictions are needed on union and intersection to guarantee that the constraints can be solved [AW93].

The function space constructor is the first example we have seen of a constructor that is not monotonic.[2] Function space is anti-monotonic in its first argument and monotonic it its second argument. That is, the following hold:

$$\begin{array}{lcll} X \to Y & \subseteq & X \to Y \cup Y' & \textit{monotonic} \\ X \to Y & \supseteq & X \cup X' \to Y & \textit{anti-monotonic} \end{array}$$

People unfamiliar with the type theory of functions often find the property of anti-monotonicity surprising. The explanation is in the definition of function space above. Note the implication in the set qualification "$x \in X \Rightarrow f(x) \in Y$". Increasing X strengthens the hypothesis, so fewer functions f satisfy the implication and the resulting set is smaller. Increasing Y weakens the conclusion, so more functions f satisfy the implication and the resulting set is larger. Function spaces are used primarily in the analysis of functional programming languages [AW93, AWL94, AF95, FA97, MW97, FFK$^+$96, FF97].[3]

[2]A function f is monotonic if whenever $x \leq y$ then $f(x) \leq f(y)$.

[3]It is also possible to define analyses involving functions that avoid anti-monotonic constructors altogether, although these techniques assume the entire program is available to be analyzed at once [Hei94, FF97].

3.2.2 Conditional Expressions

Conditional expressions $Y \Rightarrow X$ are equal to X if Y is non-empty and equal to 0 otherwise:

$$Y \Rightarrow X = \begin{cases} 0 & \text{if } Y = 0 \\ X & \text{if } Y \neq 0 \end{cases}$$

Conditional expressions are very useful for expressing constraints on flow of control in programs. For example, consider the following `case` statement on a boolean expression.

```
case x of
  true: y;
  false: z;
esac
```

We may wish to construct an analysis that captures the fact that the result of this expression can be `y` only if `x` evaluates to `true` and that the result can be `z` only if `x` evaluates to `false`. Let $[\![\cdot]\!] : \mathit{Expressions} \rightarrow \mathit{SetVariables}$ be a function mapping a program phrase to a set variable corresponding to the analysis of that phrase in the solutions of the constraints (this notation is taken from [PS91]). Assuming that `true` and `false` are set constructor constants with the obvious interpretations, then the desired constraint for the `case` expression is

$$(([\![\mathtt{x}]\!] \cap \mathtt{true}) \Rightarrow [\![\mathtt{y}]\!]) \cup (([\![\mathtt{x}]\!] \cap \mathtt{false}) \Rightarrow [\![\mathtt{z}]\!]) \subseteq [\![\mathtt{case\ x\ of\ true:\ \ y;\ false:\ \ z;\ esac}]\!]$$

It is worthwhile noting that from the point of view of decidability, conditional expressions add nothing to set constraints as they are a special case of projections. To see this, observe that

$$Y \Rightarrow X \equiv c^{-1}(c(X, Y))$$

Here we rely on the fact that the interpretation of constructors requires that if $Y = 0$, then $c(X, Y) = 0$ for any X. If one wishes to compute solutions (and not just know that solutions exist), then it turns out that for a language without explicit projections but with conditional expressions it is possible to finitely represent all solutions of the constraints [AWL94].

We shall sometimes find it convenient to allow conditional constraints in addition to conditional expressions. A conditional constraint has the form

$$X \Rightarrow (Y \subseteq Z)$$

and has the meaning that if $X \neq 0$ then $Y \subseteq Z$ must hold and otherwise there is no constraint. Conditional expressions and conditional constraints are equivalent in the sense that

$$X \Rightarrow (Y \subseteq Z) \equiv (X \Rightarrow Y) \subseteq Z$$

4 Applications

This section presents applications of set constraints to three classical program analysis problems: dataflow analysis, type inference, and closure analysis. We expect that at least one of the chosen applications is familiar to any reader with a background in one of the major program analysis communities. We use set constraints as the common language in which the analysis problems are presented.

4.1 Dataflow Analysis

Classical dataflow computations for imperative languages include live variable analysis, reaching definitions, and constant propagation, among others [ASU86]. These algorithms are formalized as the solution of systems of constraints over expressions built from sets of constants, set variables, and the set operations:

$$E ::= a_1 \mid \ldots \mid a_n \mid \alpha \mid E_1 \cap E_2 \mid E_1 \cup E_2 \mid \neg E_1$$

In this grammar $a_1, \ldots, a_n$ are the constants (nullary constructors) and α stands for a family of set variables. The meaning of an expression is a set of constants. A system of constraints is a conjunction of equalities $\bigwedge_i \alpha_i = E_i$. We assume that each variable appears on the left-hand side of at most one equation.

For example, in a live variable analysis in a language such as FORTRAN there is one constant for each program variable. The problem is to compute, for each program statement S, the variables x that may be used after the execution of S without any intervening assignments to x. For brevity we consider only the case where S is an assignment statement; the formulation for other program constructs is also straightforward. For each assignment statement we need to know two constant sets:

- S_{def} is the set of variables defined (written) by S.
- S_{use} is the set of variables used (read) by S.

For example, in the statement $\mathtt{x} = \mathtt{x} + \mathtt{y}$ we have $S_{def} = \mathtt{x}$ and $S_{use} = \mathtt{x} \cup \mathtt{y}$. For each statement S there are two set variables $[\![S]\!]_{in}$ and $[\![S]\!]_{out}$, corresponding to the set of variables live immediately before and after S respectively. Let $succ(S)$ be the statements immediately after S in program execution. The system of constraints is then

$$\begin{aligned} [\![S]\!]_{in} &= S_{use} \cup ([\![S]\!]_{out} \cap \neg S_{def}) \\ [\![S]\!]_{out} &= \bigcup_{X \in succ(S)} [\![X]\!]_{in} \end{aligned}$$

These constraints express how live variables are (or are not) propagated from one program statement to another. For example, for the statement $\mathtt{x} = \mathtt{x} + \mathtt{y}$ the first constraint is

$$[\![S]\!]_{in} = \{x, y\} \cup ([\![S]\!]_{out} \cap \neg\{x\})$$

which is equivalent to

$$[\![S]\!]_{in} = \{x, y\} \cup [\![S]\!]_{out}$$

There are a few subtleties in our formulation of live variable analysis worth discussing. First, note the optimization of the constraint representation in the immediately preceding lines (i.e., where an intersection is eliminated from the right-hand side of the equation). In the process of solving the equations it may be necessary to evaluate individual equations many times under different assignments to the variables. Thus, applying identities to simplify constraints can significantly improve the performance of constraint resolution implementations. This example merely hints at what transformations are possible, and there is a substantial literature on simplifying set constraints [Pot96, TS96, FA96, FF97, MW97].

Second, we have actually stretched the truth and presented a significant generalization of the classical dataflow theory. Note that the set expression grammar above allows negation of arbitrary expressions $\neg E$. The standard proof that dataflow equations have solutions requires that all operators be monotonic, which $\neg$ clearly is not. To achieve monotonicity, set complement is restricted to statically known sets (i.e., set expressions without variables) in which case the right-hand sides of equations are monotone in all variables. This restriction is not strictly required—the constraints presented (with $\neg$) can be solved as they are a special case of more general set constraints for which resolution algorithms are known [AW92].

There are reasons, however, to prefer restricted set complement in dataflow analysis. First, adding general complement raises the computational complexity significantly (see discussion at the end of this section). Second, in dataflow analysis we usually are interested in a best solution, either the least or the greatest. A unique best solution need not exist if set complement is unrestricted. For the purposes of dataflow analysis, we shall assume simply that negation is used in a such a way that set expressions are monotone in all variables.

For live variable analysis it is the least solution that is desired. In this case, the following inclusion constraints are equivalent:

$$\begin{aligned}
[\![S]\!]_{in} &\supseteq S_{use} \cup ([\![S]\!]_{out} \cap \neg S_{def}) \\
[\![S]\!]_{out} &\supseteq \bigcup_{X \in succ(S)} [\![X]\!]_{in}
\end{aligned}$$

As a useful exercise in manipulating constraints we now show that these inclusions have the same least solution as the equalities. (Solution θ is least if for any other solution θ', we have $\theta(x) \subseteq \theta'(x)$ for all x.) Because equality implies inclusion, it follows that every solution of the equalities is also a solution of the inclusions. Therefore, it suffices to show that the inclusions have a least solution that is also a solution of the equations.

As a first step, note that the constraints always have a solution $\alpha_i = \{a_1, \ldots, a_n\}$ (the set of all constants). Every inclusion constraint is satisfied because the left-hand side is the largest possible set.

Let θ_1 and θ_2 be any solutions of the inclusions and let $\theta_3(\alpha) = \theta_1(\alpha) \cap \theta_2(\alpha)$. Now for every inclusion constraint $\alpha \supseteq E$ we have

$$\begin{array}{lllll}
\theta_1(\alpha) & \supseteq & \theta_1(E) & \supseteq & \theta_3(E) \\
\theta_2(\alpha) & \supseteq & \theta_2(E) & \supseteq & \theta_3(E)
\end{array}$$

where the last step of both lines follows by monotonicity. It follows that

$$\theta_1(\alpha) \cap \theta_2(\alpha) = \theta_3(\alpha) \supseteq \theta_3(E)$$

so θ_3 is also a solution of the inclusions. Since there always exists a solution, solutions are closed under intersection, and there are only finitely many solutions (because the domain is finite and there are a finite number of variables), there must be a least solution.

Let θ be the least solution of the inclusions and assume for the sake of a contradiction that it is not a solution of the equalities. Then there is a constraint $\alpha \supseteq E$ such that $\theta(\alpha) \supset \theta(E)$. Let $\theta' = \theta[\alpha \leftarrow \theta(E)]$. Now we have

$$\theta(\alpha) \supseteq \theta'(\alpha) = \theta(E) \supseteq \theta'(E)$$

by monotonicity. For any other constraint $\alpha' \supseteq E'$ we know $\alpha \neq \alpha'$ (recall every variable appears in at most one left-hand side), and we have

$$\theta(\alpha') = \theta'(\alpha') \supseteq \theta'(E')$$

where the last $\supseteq$ again follows by monotonicity. Thus, θ' is a solution smaller than θ, a contradiction. We conclude that θ is a solution of the equalities.

Dataflow equations are a special case of set constraints where the only constructors are constants, the left-hand side of an equation is always a variable, and set complement is restricted. The decidability of these equality constraints follows immediately from the decidability of set constraints. More interestingly, though, the decidability of extensions also follows immediately. As noted above, unrestricted complement can be added and all solutions are still computable, although the computational complexity increases from polynomial time to NP-complete [AKVW93].

Two other set constraint extensions to dataflow analysis are particularly useful. The first is the addition of conditional expressions $X \Rightarrow Y$. As noted earlier, conditional expressions can be used to model control flow, which complements the emphasis on data flow in (aptly named) dataflow analysis. A good example of the combination of these features is found in [Hei94, AFS98]. The second extension is the ability to perform dataflow analysis of data structures by including non-atomic constructors. Set-based analysis is a canonical example of a system that exploits this feature of set constraints [Hei92, Hei94].

Finally, the algorithm given by the constraint resolution rules is unlikely to be as efficient as the standard algorithms for live variable analysis. The culprit is the rule for adding transitive constraints

$$E_1 \subseteq \alpha \wedge \alpha \subseteq E_2 \equiv E_1 \subseteq \alpha \wedge \alpha \subseteq E_2 \wedge E_1 \subseteq E_2$$

which adds new constraints between variables $\alpha \subseteq \beta \subseteq \gamma \Rightarrow \alpha \subseteq \gamma$, something that practical implementations for this problem do not do. To achieve an algorithm with efficiency akin to those used in practice, we can modify the rule for transitive constraints to propagate only constants in lower bounds to upper bounds:

$$a \subseteq \alpha \wedge \alpha \subseteq E \equiv a \subseteq \alpha \wedge \alpha \subseteq E \wedge a \subseteq E$$

It is easy to show that this rule makes the least solution explicit; each variable is assigned the set of constants appearing in its lower bound.

4.2 Simple Type Inference

Type inference is a central component of statically typed functional languages. The essence of the inference algorithm is to generate a system of type constraints from the program text. If the constraints are solvable then the program is typable and the types of program phrases are exhibited by the solutions of the constraints.

For our purposes the pure lambda calculus suffices as the programming language:

$$e ::= x \mid \lambda x.e_1 \mid e_1\, e_2$$

For simplicity, we assume that variables in an expression are renamed as necessary so that all lambda bound variables are distinct. For a simple (that is, not polymorphic) type system, the expressions of the constraint language are

$$E ::= \alpha \mid E_1 \rightarrow E_2$$

where $\rightarrow$ is an infix binary type constructor. Constraint systems are conjunctions of equations $\bigwedge_i E_{i1} = E_{i2}$. As discussed in Section 3.2.1, the term model presented in Section 3 is inadequate for function spaces, but adequate models do exist.

There are many equivalent ways to specify simple type inference. One which is close to actual implementations of type inference algorithms uses systems of type equations. As before, we use $[\![e]\!]$ to stand for a type variable associated with e.

$$\begin{array}{rcl} [\![\lambda x.e]\!] & = & [\![x]\!] \to [\![e]\!] \\ [\![e_1]\!] & = & [\![e_2]\!] \to [\![e_1\ e_2]\!] \end{array}$$

This formulation is equivalent to the standard one which uses inference rules and is well-known [Wan87]. Under these rules it is easy to verify the types of the following examples:

$$\begin{array}{rcl} \lambda x.x & : & \alpha_x \to \alpha_x \\ \lambda z.\lambda y.z & : & \alpha_z \to (\alpha_y \to \alpha_z) \\ (\lambda z.\lambda y.z)\lambda x.x & : & \alpha_y \to (\alpha_x \to \alpha_x) \\ \lambda f.\lambda x.f(f(x)) & : & (\alpha_x \to \alpha_x) \to \alpha_x \to \alpha_x \end{array}$$

Depending on whether finite or infinite solutions are desired, the constraints are solved using respectively unification or circular unification. If circular unification is used, then every lambda expression has a type. (To see this, note that both equations can be solved by assigning every expression the recursive type $\alpha = \alpha \to \alpha$.) Not every expression has a type using ordinary unification. Of course, an alternative proof of decidability is to observe that these are set constraints. Note, however, that just as in the case of unification an occurs check is required if only finite solutions are desired.

4.3 A Variation

Once again we can obtain generalizations of the familiar theory. For example, by generalizing terms to sets we can define the following grammar for types:

$$E = \alpha \mid E_1 \to E_2 \mid E_1 \cap E_2 \mid E_1 \cup E_2 \mid 0$$

We recast the constraints to use inclusion instead of equality and allow solutions to be expressed in terms of the more expressive types:

$$\begin{array}{rcl} [\![\lambda x.e]\!] & \supseteq & [\![x]\!] \to [\![e]\!] \\ [\![e_1]\!] & \subseteq & [\![e_2]\!] \to [\![e_1\ e_2]\!] \end{array}$$

The first constraint says simply that the type of $\lambda x.e$ must include all the functions of type $[\![x]\!] \to [\![e]\!]$. To understand the second constraint, note that for the constraints to have any solutions $[\![e_1]\!]$ must be a set of functions. Assume $[\![e_1]\!] = X \to Y$ for some X and Y. We then have

$$[\![e_1]\!] = X \to Y \subseteq [\![e_2]\!] \to [\![e_1\ e_2]\!]$$

which implies, using the anti-monotonicity of the domain and monotonicity of the range, that

$$[\![e_2]\!] \subseteq X \wedge Y \subseteq [\![e_1\ e_2]\!]$$

In other words, the domain X of e_1 must accept the type of the argument $[\![e_2]\!]$, and the type of the result $[\![e_1\ e_2]\!]$ must be at least the range Y of e_1.

Under these inclusion constraints many functions have substantially more precise types than under the original equality constraints. For example, the function that applies a function twice to its argument has the type:

$$\lambda f.\lambda x.f(f(x)) : ((\alpha \to \beta) \cap (\beta \to \gamma)) \to (\alpha \to \gamma)$$

Note that now the function f may be overloaded. The constraints imply that the function is well-typed provided that f has signatures $\alpha \to \beta$ and $\beta \to \gamma$ that can be composed to produce a function of type $\alpha \to \gamma$.

The extended type system presented here is somewhat related to *intersection type* disciplines. The language of intersection types retains variables, function spaces, and intersections between types, but no 0 or type union. However, most intersection type disciplines have much more general rules for assigning types to expressions than the constraint generation rules we give above. As a result, even typechecking for the natural intersection type discipline is undecidable [CC90]. Restricted, decidable versions of intersection type systems have received considerable attention (see, e.g., [CG92]).

4.4 Closure Analysis

A standard program analysis for functional languages is *closure analysis.* Because closure analysis is not as well-known as dataflow analysis and type inference, we first describe a simple closure analysis before discussing constraints.

Intuitively, the closure analysis problem for the lambda calculus is to estimate the set of lambda abstractions to which a program variable can be bound during reduction. For example, in the expression $(\lambda x.x)\lambda y.y$, the variable x will be bound to an expression beginning λy, while y will not be bound to any expression. Closure analysis is used to derive an approximation of the *control flow graph* in a higher order functional language. In a first order language (such as FORTRAN) the control flow graph is statically known—the order in which expressions are evaluated is obvious from program syntax, and this order is the structure from which dataflow analysis algorithms are built. In a higher order language, the order in which expressions are evaluated must be inferred and, in general, approximated. Closure analysis is a well-known algorithm for approximating the control-flow graph of a program and has been studied extensively [Shi88, Ses91, PS91, Pal95, NN97].

Our development of closure analysis follows Palsberg's. Let $[\![e]\!]$ be a variable associated with expression e; this variable ranges over sets of lambda bindings appearing in the complete expression. For example, for the expression $\lambda x.\lambda y.x$ the set of lambdas is $\{\lambda_x, \lambda_y\}$. For a fixed lambda expression e, the closure analysis is the least solution of a system of constraints derived from the sub-expressions of e:

Sub-Expression	*Constraints*
$\lambda x.e_0$	$\lambda_x \subseteq [\![\lambda x.e_0]\!]$
$e_1\, e_2$	for every $\lambda x.e_3$ in e $\lambda_x \subseteq [\![e_1]\!] \Rightarrow ([\![e_2]\!] \subseteq [\![x]\!] \;\wedge\; [\![e_3]\!] \subseteq [\![e_1\, e_2]\!])$

For the expression $(\lambda x.x)\lambda y.y$, the constraints are

$$\begin{array}{c}
\{\lambda_x\} \subseteq [\![\lambda x.x]\!] \\
\{\lambda_y\} \subseteq [\![\lambda y.y]\!] \\
\lambda_x \subseteq [\![\lambda x.x]\!] \Rightarrow ([\![\lambda y.y]\!] \subseteq [\![x]\!] \;\wedge\; [\![x]\!] \subseteq [\![(\lambda x.x)\lambda y.y]\!]) \\
\lambda_y \subseteq [\![\lambda x.x]\!] \Rightarrow ([\![\lambda y.y]\!] \subseteq [\![y]\!] \;\wedge\; [\![y]\!] \subseteq [\![(\lambda x.x)\lambda y.y]\!])
\end{array}$$

Solutions of the constraints are ordered pointwise; i.e., $\sigma \leq \sigma'$ if and only if $\sigma(x) \subseteq \sigma'(x)$ for all x. It is easy to verify that the least solution of the constraints is

$$\begin{array}{rcl}
[\![x]\!] & = & \{\lambda_y\} \\
[\![y]\!] & = & \emptyset
\end{array}$$

$$\begin{array}{rcl} [\![\lambda x.x]\!] & = & \{\lambda_x\} \\ [\![\lambda y.y]\!] & = & \{\lambda_y\} \\ [\![(\lambda x.x)\lambda y.y]\!] & = & \{\lambda_y\} \end{array}$$

Our definition of closure analysis introduces two small extensions to the constraint notation we have defined. Define $c \subseteq X \Rightarrow P$ to mean $X \cap c \Rightarrow P$, which is equivalent but stays within our syntax. Also, define $X \Rightarrow (Y \wedge Z)$ to mean $(X \Rightarrow Y) \wedge (X \Rightarrow Z)$.

The fact that set constraints of this form can be solved for the least solution in time $\mathcal{O}(n^3)$ follows immediately from more general results on solving systems of set constraints [Hei94, AWL94] (see Section 5). Historically, however, closure analysis has been investigated over a period of many years in isolation from other techniques and, essentially, the fragment of set constraints needed for the problem has been discovered from first principles [Shi88, PS91]. Set-based analysis can be viewed as a more general form of closure analysis where, among other things, there is some ability to track the flow of control through conditional tests [Hei94].

5 Solving Constraints

So far we have worked at the level of specifying the constraints for particular program analysis applications. In this section we discuss computing solutions of constraints. The general strategy in constraint resolution algorithms is always the same: An initial system of constraints is repeatedly transformed using simple rules until the system is in a "solved form." We illustrate this approach using the three analysis problems presented in Section 4.

We begin by defining our notion of a solved form system of constraints. We show that any *inductive* system of constraints has solutions, and that in fact all solutions are explicit in the form of the constraints (Section 5.1). In the following subsections we give algorithms for transforming the constraint systems developed in Section 4 into inductive form.

5.1 Inductive Systems

We shall limit our discussion to the following expression language, which excludes projections.

$$E ::= \alpha \,|\, 0 \,|\, E_1 \cup E_2 \,|\, E_1 \cap E_2 \,|\, \neg E_1 \,|\, c(E_1, \ldots, E_{a(c)})$$

Much of the development in this section follows [AW93].

We make use of two previous results in the proof that inductive systems have solutions. The first is a technique for transforming inclusion constraints to an equivalent system of equations [AW92]. The second is the fact that systems of *contractive* equations have unique solutions [MPS84]. The constraint-solving algorithm presented in Section 5 reduces an initial system of constraints to a set of systems of inductive constraints or reports that the initial system is inconsistent.

To discuss constraint solving it is necessary to be fairly specific about the semantic domain. We have discussed two domains, a domain of terms and a domain that includes function spaces. For simplicity, we shall prove our results only for the term domain. We need the following definition. Let D_j be an increasing sequence of sets that contain larger terms (terms of greater height) as j increases:

- $D_0 = \emptyset$
- $D_j = \{c(t_1, \ldots, t_{a(c)}) | t_j \in D_{j-1}\} \cup D_{j-1}$

To help motivate the technical definitions that follow, consider the following natural inductive strategy for showing that an arbitrary system of inclusion constraints over variables $\alpha_1, \ldots, \alpha_n$ has a solution. Initially, let $\alpha_i = 0$ for $1 \leq i \leq n$. At step j of the induction, assign some terms of D_j to α_1, then to α_2, and so on, up to α_n. At each step (j, i) of this double induction over the terms of D_j and variables α_i, we must ensure that the constraints are satisfied for all elements in D_j. If this can be done for all pairs (j, i) then the system has a solution.

In such an inductive proof, we must distinguish between variables inside of constructors $c(\alpha)$, which contribute terms from D_{j-1}, and variables outside of constructors $\alpha \cap c(\ldots)$, which contribute terms from D_j.

Definition 5.1 The *top-level variables* of X (denoted $TLV(X)$) are the variables in X that appear outside of a constructor. Formally,

$$\begin{array}{rcl} TLV(\alpha_i) & = & \{\alpha_i\} \\ TLV(0) & = & \emptyset \\ TLV(c(\ldots)) & = & \emptyset \\ TLV(E_1 \cup E_2) & = & TLV(E_1) \cup TLV(E_2) \\ TLV(E_1 \cap E_2) & = & TLV(E_1) \cup TLV(E_2) \\ TLV(\neg E_1) & = & TLV(E_1) \end{array}$$

Top-level variables are also called the *non-expansive* variables [MPS84].

Definition 5.2 A system S of constraints is *inductive* if the following three conditions hold:

1. $S = \bigwedge_{1 \leq i \leq n} L_i \subseteq \alpha_i \subseteq U_i$ (i.e., there is one lower bound L_i and upper bound U_i per variable α_i)
2. $TLV(L_i) \cup TLV(U_i) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\}$ for $1 \leq i \leq n$
3. For all $i_0 = 1, \ldots, n$ and integers j, the following holds in all assignments:

   $(\forall i = 1, \ldots, i_0 - 1\ (L_i \cap D_j \subseteq \alpha_i \cap D_j \subseteq U_i \cap D_j)$ and
   $\forall i = i_0, \ldots, n\ (L_i \cap D_{j-1} \subseteq \alpha_i \cap D_{j-1} \subseteq U_i \cap D_{j-1}))$
   $\Rightarrow L_{i_0} \cap D_j \subseteq U_{i_0} \cap D_j$

Parts 1 and 2 are simple syntactic properties. Part 3 is a more complex semantic condition. The double induction outlined above for constructing solutions is expressed in part 3, which says that if the constraints are satisfiable up to some level i_0 and variable α_{j-1}, then the constraints are satisfied for the next lower and upper bound pair in the induction $L_{i_0} \cap D_j \subseteq U_{i_0} \cap D_j$.

Definition 5.2 makes it possible to build solutions inductively at level D_j by assigning values in order to $\alpha_1, \ldots, \alpha_n$ since part 2 ensures that variables are constrained only by lower-numbered variables at the top level and part 3 ensures that α_{i_0} can be given a value between L_{i_0} and U_{i_0}. Systems that do not satisfy part 3 may not have any solutions (consider, for example, system $1 \subseteq \alpha_1 \subseteq 0$).

Inductive systems are the output of our constraint resolution procedures. That is, we will give procedures (starting in Section 5.3) for transforming an initial constraint system into an equivalent system in inductive form. For these resolution algorithms we can prove that if the output of the algorithm contains no trivially inconsistent constraints (e.g., $1 \subseteq 0$ or $\texttt{int} \subseteq 0$) then the system is in inductive form and therefore has solutions.

We show that inductive systems have solutions in two steps: first, we show that an inductive system is equivalent to a system of equations; we then show that the equations always have solutions.

Definition 5.3 A system of equations $\alpha_1 = E_1 \wedge \ldots \wedge \alpha_n = E_n$ (where each α_i appears on one left-hand side) is *cascading* if $TLV(E_i) \cap \{\alpha_i, \ldots, \alpha_n\} = \emptyset$.

Theorem 5.4 Let $S = \bigwedge_i L_i \subseteq \alpha_i \subseteq U_i$ be an inductive system of constraints. Then S is equivalent to the cascading equations $\alpha_i = L_i \ \cup \ (\beta_i \cap U_i)$ where the β_i are fresh variables.

Proof: Assume that $L_i \subseteq \alpha_i \subseteq U_i$ and let $\beta_i = \alpha_i$. Then

$$\begin{aligned} \alpha_i &= L_i \cup (\alpha_i \cap U_i) && \text{since } L_i \subseteq \alpha_i \subseteq U_i \\ &= L_i \cup (\beta_i \cap U_i) && \text{since } \alpha_i = \beta_i \end{aligned}$$

Thus, every solution of the constraints induces a solution of the equations. For the other direction, assume that $\alpha_i = L_i \cup (\beta_i \cap U_i)$ for some β_i. Clearly, $L_i \subseteq \alpha_i$. To show $\alpha_i \subseteq U_i$, we first show for all i and j that $\alpha_i \cap D_j \subseteq U_i \cap D_j$. For the sake of obtaining a contradiction, assume $\alpha_i \cap D_j \not\subseteq U_i \cap D_j$ for some i and j. Pick the smallest such pair (j, i) ordered lexicographically. Note $L_k \cap D_l \subseteq \alpha_k \cap D_l \subseteq U_k \cap D_l$ holds if $(k, l) < (j, i)$ by assumption and because $L_k \subseteq a_k$. Since the system is inductive, it follows that $L_i \cap D_j \subseteq U_i \cap D_j$. Therefore

$$\begin{aligned} & \alpha_i \cap D_j \\ = \ & (L_i \cup (\beta_i \cap U_i)) \cap D_j \\ = \ & (L_i \cap D_j) \cup (\beta_i \cap U_i \cap D_j) \\ \subseteq \ & U_i \cap D_j \end{aligned}$$

which contradicts the assumption. Thus for all i,

$$\begin{aligned} & \alpha_i \cap D_j \subseteq U_i \cap D_j && \text{for all } j \\ \Rightarrow \ & \alpha_i \cap D_j \subseteq U_i && \text{for all } j \\ \Rightarrow \ & \alpha_i \subseteq U_i && \text{since } \textstyle\bigcup_j D_j = H \end{aligned}$$

□

Theorem 5.5 shows that every choice for the β_i induces a unique solution to the cascading equations.

Theorem 5.5 Let $\alpha_1 = E_1 \wedge \ldots \wedge \alpha_n = E_n$ be a system of cascading equations and let σ be any assignment for the variables other than the $\{\alpha_1, \ldots, \alpha_n\}$. There is a unique extension σ' of σ that is a solution of the equations.

Proof: Variable α_i can be eliminated from the top-level variables of every equation by substituting E_i for α_i in E_{i+1} through E_n. Let β be any remaining top-level free variable. Then β does not appear on the left-hand side of any equation; we call such variables *free*. For any fixed assignment σ for the top-level free variables, the equations become *contractive* (have no top-level variables). Contractive equations have unique solutions [MPS84]. □

5.2 A Digression on Set Complement

Set complement is quite handy for expressing analyses, but in solutions of constraints we often wish to eliminate complements so that we can see which terms may belong to an expression E rather than which terms may not belong to E. The following identities are used to drive complements inwards in the cascading equations:

$$\begin{aligned}
\neg 0 &= 1 \text{ where } 1 = \bigcup_{c\in C} c(1,\ldots,1) \\
\neg(E_1 \cup E_2) &= \neg E_1 \cap \neg E_2 \\
\neg(E_1 \cap E_2) &= \neg E_1 \cup \neg E_2 \\
\neg\neg E &= E \\
\neg c(E_1,\ldots,E_{a(c)}) &= c(\neg E_1, 1,\ldots,1) \cup \ldots \cup c(1,\ldots,1,\neg E_{a(c)}) \cup \bigcup_{d\in C-\{c\}} d(1,\ldots,1)
\end{aligned}$$

The equation in the first line defines 1 to be the Herbrand universe. For each equation $\alpha_i = E_i$ create a new equation $\neg\alpha_i = \neg E_i$ and simplify the right-hand side.[4] Now replace $\neg\alpha_i$ everywhere by a fresh variable γ_i. The preceding rules and this technique for eliminating $\neg\alpha_i$ remove all negations except on a free variable β. A negation $\neg\beta$ cannot be removed, as the β are free variables in the constraints.

There is another important issue with set complement. We have assumed that the set of constructors is finite, and therefore $\neg c(\ldots)$ can be written as above using an explicit union of all non-c terms. However, in many applications it is unreasonable to assume that we know all of the constructors. Typically the set of constructors is determined by the program text. Because a constructor defined in one part of a program potentially appears in the solutions of the constraints of any part of that program, assuming that all constructors are known at the outset makes it impossible to analyze program components separately.

It is not difficult to remove the assumption that all constructors are known. Assume now that C is an infinite set of constructors. We add the following new set expression with the semantics:

$$\sigma(NOT(\{c_1,\ldots,c_n\})) = \{d(t_1,\ldots,t_{a(d)}) | t_i \in H \wedge d \in C - \{c_1,\ldots,c_n\}\}$$

Intuitively NOT is the set of all terms with a head constructor not in the argument list. It is straightforward to include NOT in the algebra of set expressions. For example:

$$\begin{aligned}
\neg NOT(\{c_1,\ldots,c_n\}) &= c_1(1,\ldots,1) \cup \ldots \cup c_n(1,\ldots,1) \\
\neg c(E_1,\ldots,E_n) &= c(\neg E_1, 1,\ldots,1) \cup \ldots \cup c(1,\ldots,1,\neg E_n) \cup NOT(\{c\}) \\
NOT(\{c_1,\ldots,c_n\}) \cap NOT(\{d_1,\ldots,d_m\}) &= NOT(\{c_1,\ldots,c_n\} \cup \{d_1,\ldots,d_m\}) \\
1 &= NOT(\emptyset)
\end{aligned}$$

Even in the case where all constructors are known, $NOT(\{c\})$ is a more efficient representation than an explicit union of all constructors except c.

5.3 Closure Analysis

We now turn to algorithms for solving constraints. Constraint resolution is done by applying a set of rewrite rules repeatedly until closure. For pedagogical reasons we present the rules a few at a time, as needed for each application. However, it is emphasized that in developing new applications it is usually unnecessary to invent new rules. New analyses generally are expressed using the established machinery (the complete set of rules), which means the analysis designer can simply write the necessary constraints and be assured the constraints can be solved.

[4]This step only works because the cascading equations are already contractive in the α_i. For example, starting with $\alpha = \alpha$ and adding complements gives us an equation with exactly the same solutions $\neg\alpha = \neg\alpha$.

$$\begin{aligned}
S \wedge 0 \subseteq E &\equiv S & (1)\\
S \wedge E_1 \cup E_2 \subseteq E_3 &\equiv S \wedge E_1 \subseteq E_3 \wedge E_2 \subseteq E_3 & (2)\\
S \wedge \alpha \subseteq \alpha &\equiv S & (3)\\
S \wedge E_1 \subseteq \alpha \wedge \alpha \subseteq E_2 &\equiv S \wedge E_1 \subseteq \alpha \wedge \alpha \subseteq E_2 \wedge E_1 \subseteq E_2 & (4)\\
S \wedge \lambda_x \in \alpha \Rightarrow E_1 \subseteq E_2 \wedge \lambda_x \subseteq \alpha &\equiv S \wedge E_1 \subseteq E_2 \wedge \lambda_x \subseteq \alpha & (5)
\end{aligned}$$

Figure 1: Rules for simplifying constraints.

We begin with closure analysis as it has the simplest resolution procedure. Expressions have the form

$$E ::= \lambda_x \,|\, \alpha \,|\, 0 \,|\, E_1 \cup E_2 \,|\, \lambda_x \subseteq \alpha \Rightarrow E_1$$

and a system S of constraints has the form

$$S = \bigwedge_i E_i \subseteq \alpha_i$$

We say two systems are equivalent $S_1 \equiv S_2$ if they have the same set of solutions. Figure 1 gives a number of equivalences for closure analysis constraints. It is easy to verify that these are in fact equivalences.

A constraint $\alpha_i \subseteq U$ (respectively $L \subseteq \alpha_i$) is *inductive* if $TLV(U)$ (respectively $TLV(L)$ is a subset of $\{\alpha_0, \ldots, \alpha_{i-1}\}$. The algorithm for solving the closure analysis constraints is as follows.

Read the equivalences as rewrite rules going from left to right. The rules are applied to the constraint system repeatedly, in any order, until no new inductive constraints can be added.

Let S' be the result of closing the system S under the rewrite rules. The following statements are easily verified:

- $S' \equiv S$, since S' is obtained from S by a sequence of $\equiv$-preserving steps.
- There are no constraints $\lambda_x \subseteq \lambda_y$, since no constant upper bounds appear in the initial constraints and none are added by the rules.
- All constraints in S' are of the form $\alpha \subseteq \beta$, $\lambda_x \subseteq \beta$, or $\lambda_x \in \alpha \Rightarrow E_1 \subseteq E_2$. To see this, note the previous point and that all other forms of left-hand sides are eliminated by the rules.
- The procedure terminates, because constraints on the right-hand sides of the rules involve only pairs of subexpressions of the original system. There are only finitely many such pairs, so eventually no new inductive constraints can be added. To help detect when all inductive constraints have been added it is sufficient to apply the transitive rule (4) once only for each pair of inductive upper and lower bounds on a variable. With that restriction the algorithm terminates exactly when no rules apply. (Note that rules (3) and (4) cannot get into a loop because $\alpha \subseteq \alpha$ is not an inductive constraint.)

The last point can be used to perform complexity analysis of the algorithm. If the size of the original system of constraints printed as a string is n, then the size of the final system may be $\mathcal{O}(n^2)$ with $\mathcal{O}(n^2)$

constraints. Rules 1-3 involve only a single constraint and take constant time, so the total cost of these rules is $\mathcal{O}(n^2)$. For Rule 4, a variable α may have $\mathcal{O}(n)$ upper and lower bounds. Forming all pairs of upper and lower bounds for α takes $\mathcal{O}(n^2)$ time. Since there may be $\mathcal{O}(n)$ variables the total cost is $\mathcal{O}(n^3)$. The cost of Rule 5 can similarly be shown to be $\mathcal{O}(n^3)$, so the total cost is $\mathcal{O}(n^3)$.

It remains to show that the rules actually solve the constraints. From the discussion above we know that there can be no trivially inconsistent constraints of the form $\lambda_x \subseteq \lambda_y$ where $x \neq y$. Thus, when the algorithm terminates successfully all constraints are inductive.

Index the variables $\alpha_1, \alpha_2, \ldots$. We say that a constraint $y \subseteq \alpha_j$ is a *lower bound on* α_j if $y = \lambda_x$ or $y = \alpha_i$ and $i < j$. A constraint $\alpha_j \subseteq y$ is an *upper bound on* α_j if $y = \lambda_x$ or $y = \alpha_i$ and $i < j$. Now define

$$\begin{aligned} L_i &= \bigcup \{y | y \subseteq \alpha_i \in S' \text{ is a lower bound on } \alpha_i\} \\ U_i &= \bigcap \{y | \alpha_i \subseteq y \in S' \text{ is an upper bound on } \alpha_i\} \end{aligned}$$

The L_i and the U_i simply combine all upper and lower bounds on variables into a single upper and lower bound per variable. Note that the L_i and U_i exclude any conditional constraints remaining in S'.

Lemma 5.6 The system $\bigwedge_i L_i \subseteq \alpha_i \subseteq U_i$ is inductive.

Proof: Conditions (1) and (2) of Definition 5.2 are easily verified; for (2), simply note that each constraint is inductive. For condition (3), because our domain is a set of constants λ_x the hierarchy of D_i's collapses to $D_0 = \emptyset$ and $D_1 = \{\lambda_x | x \text{ is a program variable}\}$. The condition for inductiveness can then be simplified:

$$\forall 1 \leq i_0 \leq n. \forall 1 \leq i < i_0. L_i \subseteq \alpha_i \subseteq U_i \Rightarrow L_{i_0} \subseteq U_{i_0}$$

The proof is by induction on i_0. For the base case, there are no variables with index lower than α_1, so no variables can appear in L_1 or U_1. In addition U_1 contains no conditional constraints or constants (see discussion above). It follows that $U_1 = \bigcap \emptyset$, which is the entire domain, so $L_1 \subseteq U_1$ in any assignment.

For the inductive case, let θ be an assignment to the variables and assume that $\theta(L_i) \subseteq \theta(\alpha_i) \subseteq \theta(U_i)$ for all $i < i_0$. Let l be a disjunct of L_{i_0} and let u be any conjunct of U_{i_0}. Then $l \subseteq u \in S'$ by Rule 4 or the constraint is a trivial one $\alpha \subseteq \alpha$ removed by Rule 3. Assume $l \subseteq u$ is a non-trivial constraint. If either l or u is a variable its index is less than i_0. Therefore, $\theta(l) \subseteq \theta(u)$ by the induction hypothesis. Since l and u were chosen arbitrarily from L_{i_0} and U_{i_0}, it follows that $L_{i_0} \subseteq U_{i_0}$.

□

Let S''' be S' with remaining conditional constraints removed. Lemma 5.6 shows that S'' has solutions given by the equations

$$\alpha_i = L_i \cup (\beta_i \cap U_i)$$

where the β_i are fresh variables. Since all operations are monotonic,[5] the smallest of these solutions is

$$\alpha_i = L_i$$

where all $\beta_i = 0$. This solution is θ where

$$\theta(\alpha_i) = \{\lambda_x | \lambda_x \text{ appears in } L_i\}$$

[5]All operations are monotonic because we designed the constraint language to avoid negations. However, note that this is the only place monotonicity is used, and that it is used to show the existence of a least solution.

To show that our constraint resolution algorithm is sound it remains to show that S has a solution. We claim that θ is a solution of S' and therefore a solution of S. It suffices to show that

$$\theta(\lambda_x \subseteq \alpha_i) \Rightarrow \theta(E_1 \subseteq E_2)$$

is satisfied for the constraints $\lambda_x \subseteq \alpha_i \Rightarrow E_1 \subseteq E_2$ in S' but not in S''. Assume for the sake of obtaining a contradiction that $\lambda_x \subseteq \theta(\alpha_i)$. The λ_x appears in L_i. But then the hypothesis of Rule 5 is satisfied, contradicting the assumption that S is closed under the rewrite rules. We conclude that $\lambda_x \not\subseteq \theta(\alpha_i)$, so the constraint is satisfied.

5.4 Dataflow Analysis

The dataflow analysis discussed in Section 4.1 allows general set complement. Here we restrict our attention to solving the specific form of constraints arising in the live variable analysis, which do not make essential use of set complement and are therefore much easier to solve.

The universe H is a finite set of constants $a_1, a_2, \ldots, a_n$. For any set of constants A, the set expression $\neg(\bigcup A)$ can be written without a negation as $\bigcup(H - A)$. Recall the liveness constraints from Section 4.1.

$$\begin{aligned} [\![S]\!]_{in} &\supseteq S_{use} \cup ([\![S]\!]_{out} \cap \neg S_{def}) \\ [\![S]\!]_{out} &\supseteq \bigcup_{X \in succ(S)} [\![X]\!]_{in} \end{aligned}$$

The only expression not already treated in the resolution rules of Figure 1 is $\alpha \cap \neg A$, where A is a union of constants. To handle this case, we make use of the identity $X \subseteq Y \cup Z \equiv X \cap \neg Z \subseteq Y$. Three cases involving variables and constants on the left-hand side are treated separately:

$$\begin{aligned} S \wedge \alpha_i \cap A \subseteq \alpha_j &\equiv S \wedge \alpha_i \subseteq \alpha_j \cup \neg A \quad i \neq j \\ S \wedge \alpha_i \cap A \subseteq \alpha_i &\equiv S \\ S \wedge a \subseteq \alpha_i \cup A &\equiv S \wedge a \cap \neg A \subseteq \alpha_i \end{aligned}$$

The first rule works either left-to-right or right-to-left. Only one direction, however, can result in a constraint in inductive form (i.e., with the higher-numbered variable isolated). Thus, if $i > j$ the rule is applied left-to-right and if $i < j$ the rule is applied right-to-left. If $i = j$ the constraint is eliminated (the second rule). Finally, if the left-hand side is a constant a, then $a \cap \neg A$ is formed to isolate the variable on the right-hand side (the third rule). The expression $a \cap \neg A$ is simplified to either a if $a \not\subseteq A$ or 0 if $a \subseteq A$.

Adding these rules to those of Figure 1 to handle the new expression $\alpha \cap A$ is all that is required to obtain an effective algorithm. The proof of Lemma 5.6 can be applied to this extension by noting that the new rules put constraints in a form satisfying condition (2) of Definition 5.2, and that the proof that conditions (1) and (3) are satisfied is unchanged.

5.5 Simple Type Inference

The constraints for simple type inference introduce one additional form of expression $E_1 \rightarrow E_2$. The corresponding resolution rule is well-known:

$$E_1 \rightarrow E_2 \subseteq E_3 \rightarrow E_4 \equiv E_3 \subseteq E_1 \wedge E_2 \subseteq E_4 \qquad (6)$$

The antimonotonicity of the domain and the monotonicity of the range are reflected in the constraints on the right-hand side (see the discussion in Section 3.2). This rule can be combined with the preceding ones to give a method for solving the typing constraints. Resolution of the constraints is again in $\mathcal{O}(n^3)$ time.

The justification for this rule is outlined in Section 3.2.1. A full formalization requires considerable additional machinery from denotational semantics and is outside the scope of this paper.

6 Discussion

We now turn to the relationship of constraint-based analysis to other approaches to program analysis and its place in the theory of abstract interpretation. The accepted intellectual framework for designing and justifying program analysis algorithms is *abstract interpretation*, due to Cousot and Cousot [CC77]. Abstract interpretation treats a program analysis as a sound approximation to the exact meaning of a program. More precisely, an abstract interpretation gives a non-standard interpretation of the program that is consistent with the standard interpretation. Let $(D, \leq_D)$ and $(A, \leq_A)$ be partially ordered domains and let $\alpha : D \rightarrow A$ and $\gamma : A \rightarrow D$ be functions that form a *Galois connection*:

$$\forall d \in D, a \in A \;\; \alpha(d) \leq_A a \Leftrightarrow d \leq_D \gamma(a)$$

Then $\alpha(d)$ is the *abstraction* of d and $\gamma(a)$ is the *concretization* of a.

By defining the abstract domain A and explicit mappings α and γ it becomes possible to state precisely what it means for an abstraction of a program to be correct. For example, let P be a program with standard semantics $\mu : \mathit{Program} \rightarrow D \rightarrow D$. Let ϕ be a program analysis (an abstract interpretation) with functionality $\phi : \mathit{Program} \rightarrow A \rightarrow A$. The ϕ is a *sound* abstraction if it satisfies:

$$\forall x \in D.(\mu \; P \; x) \leq_D \gamma(\phi \; P \; \alpha(x))$$

Thus, the abstraction $\phi(P)$ conservatively models the behavior of P.

There is confusion in the literature over the meaning of the term "abstract interpretation," which is used at least to mean either a semantic framework for reasoning about program analysis (sketched above) or a particular set of techniques for constructing program analyses. The author prefers to use the term to refer to the semantic framework only. Given that meaning, abstract interpretation provides a clear, well-defined framework for proving that a program analysis is correct. We are unaware of any program analysis that cannot be explained in this framework,[6] including constraints, although we have left the abstraction and concretization functions implicit in our examples.

Program analysis is technically difficult and at the same time new problems typically bear some resemblance to older, better understood problems. Hence, there is little enthusiasm for inventing program analyses from first principles in every instance, and people have naturally developed sets of techniques that can be reused. A few of these paradigms have developed large followings. We discuss three: finite lattice methods, type inference, and constraints.

6.1 Finite Lattice Methods

One of the most popular paradigms appeared in the Cousots' seminal paper on abstract interpretation [CC77]. Program analyses in this style are variations on a theme. A finite abstract domain A is designed

[6] *Widening/narrowing* can be defined without reference to abstraction (see [CC92]). However, when used on an abstract domain there are associated abstraction and concretization functions.

(A is generally a lattice), and the program analysis is expressed as a system of recursive equations of the following 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 each σ_i is a monotonic function with signature $A^{|X|} \rightarrow A$. It is well-known that a generic iterative fixed point algorithm computes the least solution of such equations [CC77].

Given that one can design a correct analysis in this framework, the implementation is straightforward and has two additional useful properties: first, the computed analysis is the best possible within the chosen parameters (i.e., it is the least solution of the equations) and second, the analysis is guaranteed to terminate. Analyses for C and FORTRAN programs based on dataflow equations are classic examples of this program analysis paradigm.

The cookbook recipe "finite domains plus monotonic functions equals program analysis" has proven very popular, and there are an enormous number of applications of this excellent idea; representative examples include [Myc80, JM86, Hud87, Wad87, HY88, PBJ$^+$91]. The paradigm has become so popular that the term *abstract interpretation* is often used to mean this specific technique for program analysis rather than a general semantic framework. Pedagogically this is undesirable, as it implies that the semantic framework of abstract interpretation cannot be applied to other paradigms.

6.2 Type Inference

The Hindley/Milner type inference algorithm has recently become popular as a model for program analyses of a different sort. In this approach, a program analysis is specified as a non-standard type inference system. Typically, such systems are sets of deductive inference rules, with one rule for each syntactic form in the programming language. It is worth noting that analyses in this style have been designed that prove all sorts of facts about programs, many of which have little to do with types. Representative examples include [Hen92, TT94].

Specifying a program analysis as a formal logic corresponds nicely with the intuition that the role of program analysis is to prove facts about programs. However, the inference rules alone normally do not specify an algorithm. If the logic can prove multiple facts about a program, it is necessary to specify which fact should be computed by program analysis; that is, it is necessary to specify how the proof search is conducted. In practice, designing the logic often is only the first step and much hard work remains in coming up with an algorithm and analyzing its complexity. For example, implementations of Milner's type system are based on solving systems of equality constraints using unification [Rob65].

6.3 Constraints

In 1987 Wand wrote a short paper on the Hindley-Milner type system in which he proposed to recast the usual typing rules with explicit equality constraints as side conditions, which simplifies the understanding of Hindley-Milner type inference algorithms [Wan87]. This paper is apparently the first to explicitly put forth the constraint-based viewpoint (excepting Reynold's much earlier paper [Rey69]). Further development has continued to emphasize the problems of constraint resolution over the problems of deductive inference. Note that the constraint-based analysis notation for traditional type inference problems deftly avoids using inference rules at all (see Section 4.2)!

A thesis of this paper is that constraint-based analysis unifies much of the traditional dataflow views and the type inference views of program analysis. To the degree that dataflow equations are a proxy for more general abstract interpretations over finite lattices there is considerable evidence for this thesis. In the extreme, systems of equations of the form above $x_1 = \sigma_1(X)\ \ldots\ x_n = \sigma_n(X)$ can be viewed as just another system of constraints to be solved. However, this level of generality obscures several important differences.

What we refer to as finite lattice methods generally exploit three assumptions: first, a particular solution (the least or the greatest) to the equations is desired; second, the abstract functions can be arbitrary monotonic functions; and third, that a finite domain of abstract values gives sufficient precision for all programs.[7]

With respect to the first point, in constraint-based analysis a common (but not universal) view is to compute *all* solutions of the constraints. For example, the constraint resolution procedure for live variable analysis in Section 5 does not resemble the one in textbooks precisely because it computes all, rather than the least, solution of the constraints. Computing all solutions becomes necessary for separate analysis of programs split across multiple files (where the least solution of the constraints for a particular file may have little to do with the least solution of the entire program) and when there is no least solution (e.g., in the presence of anti-monotonic constructors like function space).

The second important difference lies in the nature of the abstractions chosen in finite lattice and in constraint-based analyses. All commonly used, and very nearly all proposed, finite lattice methods are either *forwards* (information flows from inputs to outputs) or *backwards* (information flows from outputs back towards inputs; live variable analysis is an example). The dataflow analyses tend to use abstract functions to represent function values. Thus, information can flow easily only in the direction of the abstract function, which is either forwards or backwards. Constraint resolution, however, naturally allows information to flow in either or both directions, allowing forwards and backwards information flow to be used in the same analysis.

It is important to understand that allowing bidirectional information flow is not a unique property of constraints. For example, the technique of *chaotic iteration* admits analyses that are neither forwards nor backwards [CC78].

The third important difference is that constraints can easily work over infinite domains, while the finite lattice methods work with a finite domain. Finite domains are a good fit for some problems (e.g., the two point domain commonly used in strictness analysis [Myc80]), but for others (e.g., particularly problems involving recursive data structures) it is more natural to work directly with an infinite domain. A problem with infinite domains, however, is that termination of the program analysis is not automatically guaranteed. In the case of set constraints the termination of constraint resolution is guaranteed; resolution computes a finite representation of the solutions of constraints over an infinite domain.

The distinction between infinite and finite domains is subtler than we have indicated. If an analysis terminates for all programs, then clearly there is finite structure (i.e., the finite computation) regardless of the choice of domain. Thus, even if the intended domain is infinite, for each program it should be possible to substitute a finite domain that behaves indistinguishably from the infinite domain.[8] Essentially this observation is used in [CC95] in showing the equivalence of several different approaches to formulating program analyses over finite and infinite domains.

Even if infinite domains can be treated using finite equivalents (as they must be if we wish to have terminating program analyses), that does not mean that infinite domains serve no useful role. In many cases an infinite domain is simply the natural framework, while the equivalent finite domain may be difficult to discover and justify. In the case of set constraints, the finite domain can be taken to be all subsets of the constraints of the initial system plus and those added by resolution rules. The full set is only discovered by solving the constraints. A similar perspective is set forth in [CC92] in another

[7]Or that a suitable finite domain can be derived from each particular program.

[8]Note that there may be a different finite domain for each possible input program.

discussion of finite vs. infinite domains.

No discussion of infinite domains is complete without mentioning the use of *widening* to achieve termination in infinite abstract domains. Widening is very general and can be applied in any domain, finite or infinite [CC92]. Widening has two drawbacks, however. First, the price for generality is that widening is not guaranteed to produce a best solution. Second, widening is defined operationally (in terms of how it accelerates convergence). Both of these properties are undesirable in applications where users must be able to understand the results of the analysis and, if necessary, how to modify their programs so that the analysis produces better results. (Type inference is the canonical example of an analysis where user understanding is a requirement.) In other applications where user involvement is not expected, such as low-level compiler optimizations, these concerns are unimportant.

6.4 Other Constraint Systems

Constraints are a popular formalism for program analysis and the associated literature is large. We give a necessarily abbreviated survey of this work.

The most widely used constraint language is undoubtedly equality constraints between terms, solved via unification (see [Ste96] for a recent example). Unification and its variants are almost the only technique where performance has been demonstrated to scale well to large programs. While we have argued that such constraints can be captured as set constraints (which they can), there is an important distinction to be made. The generic resolution algorithm for set constraints is at least $\mathcal{O}(n^3)$ while term equations can be solved in nearly linear time. Thus, straightforward set constraint algorithms are not necessarily the best implementation of any particular fragment of set constraints.

Equations between *record types* are another popular constraint formalism, intermediate in power between term equations and set constraints [Ŕ89, Wan93]. A record type is a set of typed fields. For example $\{x : \mathit{int},\ y : \mathit{int}, \rho\}$ is a record with two fields x and y, both of type *int*. In program analysis applications the "types" in a record are replaced by descriptions appropriate to the particular analysis. An important aspect of record types is that additional, unknown fields are permitted through variables that range over record extensions. In the example above, ρ may take on any set of fields and associated types except for x and y. In this way record types allow polymorphism not just over particular record fields but also over record extensions.

Missing from set constraints is the notion that constructors may stand in non-trivial inclusion relationships to each other. For example, we may have a rule that $c(X) \leq d(X)$ for any X. For the case where there are only nullary constructors (constants) and where the inclusion ordering defines a meet semi-lattice, the inclusion constraints can be solved in linear time [RM96]. The case where the inclusion relationships do not define a semi-lattice is more difficult (as shown in [RM96]; an earlier example is [Mit91]). The situation for higher-arity constructors with inclusion relationships is less clear; see [BM97] for an example of such a system.

The examples discussed so far are primarily aimed at analyzing data structure or type descriptions. A bit afield from these kinds of constraints are integer constraints, which find application in gathering information about patterns of array references and loop bounds. The studies done using the Omega system are good examples of how a well-engineered integer constraint library simplifies many tasks (see, e.g., [Pug91, PW95]).

Beyond the standard formalisms, there are a number of more specialized constraint systems that have been developed for particular analysis problems; [Hen92, TT94] are good examples. These constraint languages have specialized features that are not easily categorized.

A very important consideration in program analysis of any sort is how polymorphism (also called polyvariance and context sensitivity) is expressed. Polymorphic analysis is a large topic in its own right and beyond the scope of this paper. Constraints are well adapted to using the standard let-style polymorphism of functional languages. In some cases even more powerful polymorphic recursion can be used [Hen88, TT94].

Another approach to constraint-based analysis is to mix multiple constraint systems in a single application [FA97]. This idea has the advantage that one need no longer find a single constraint theory that models all needed aspects of a program. Instead, different aspects of computation can be modeled separately, using whatever constraints are appropriate for efficiency or semantic reasons.

7 Conclusions

As a field, program analysis suffers from a fair degree of balkanization, with several different traditions that address related problems with related techniques but different terminology, thereby obscuring what is common and what is different. We have given a brief overview of constraint-based program analysis, focusing on three classical analyses (dataflow analysis, type inference, and closure analysis) and showing how they can be presented using the constraint-based point of view. We hope these examples serve to lower the barriers to understanding between the different program analysis communities.

8 Acknowledgments

This paper is based in part on an invited talk on constraint-based program analysis given with Nevin Heintze at the ACM Symposium on Principles of Programming Languages in 1995. Much thanks goes to Nevin for discussions on the relationship of constraint-based analysis to other analysis paradigms. Jeff Foster provided many useful comments on an earlier draft of this paper and Manuel Fähndrich provided corrections to some errors in the first published version.

References

[AF95] A. Aiken and M. Fähndrich. Dynamic typing vs. subtype inference. In *Proceedings of the 8th Conference on Functional Programming and Computer Architecture*, pages 192–191, June 1995.

[AFS98] A. Aiken, M. Fähndrich, and Z. Su. Detecting races in relay ladder logic programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 4th International Conference, TACAS'98*, volume 1384 of *LNCS*, pages 184–200, Lisbon, Portugal, 1998. Springer.

[Aik94] A. Aiken. Set constraints: Results, applications, and future directions. In *Second Workshop on the Principles and Practice of Constraint Programming*, pages 171–179, Orcas Island, Washingtion, May 1994. Springer-Verlag LNCS no. 874.

[AKVW93] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic '93*, volume 832 of *Lect. Notes in Comput. Sci.*, pages 1–17. Eur. Assoc. Comput. Sci. Logic, Springer, September 1993.

[AKW95] A. Aiken, D. Kozen, and E. Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44, 1995.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AW92] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[BGW93] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Symposium on Logic in Computer Science*, pages 75–83, June 1993.

[BM97] Francois Bourdoncle and Stephan Merz. Type-checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, January 1997.

[CAC$^+$81] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via graph coloring. *Computer Languages*, 6(1):47–57, 1981.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by contruction or approximation of fixed points. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CC78] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *Formal Description of Programming Concepts.* North-Holland, 1978.

[CC90] F. Cardone and M. Coppo. Two extensions of Curry's type inference system. In P. Odifreddi, editor, *Logic and Computer Science.* Academic Press, 1990. volume 31 of APIC series.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92*, pages 269–295. Spring-Verlag, 1992. volume 631 of LNCS.

[CC95] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. *Lecture Notes in Computer Science*, 939:293–303, 1995.

[CG92] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming.* Spring-Verlag, 1992. Lecture Notes in Computer Science 581.

[CP94a] W. Charatonik and L. Pacholski. Negative set constraints wtih equality: An easy proof of decidability. In *Symposium on Logic in Computer Science*, July 1994. To appear.

[CP94b] W. Charatonik and L. Pacholski. Set constraints with projections are in NEXPTIME. In *Foundations of Computer Science*, 1994. To appear.

[FA96] M. Fähndrich and A. Aiken. Making set-constraint program analyses scale. In *CP96 Workshop on Set Constraints*, August 1996.

[FA97] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, September 1997.

[FF97] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[FFK$^+$96] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[GTT92] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 505–514, 1992.

[GTT93] R. Gilleron, S. Tison, and M. Tommasi. Solving Systems of Set Constraints with Negated Subset Relationships. In *Foundations of Computer Science*, pages 372–380, November 1993.

[Hei92] N. Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[Hei94] N. Heintze. Set-based analysis of ML programs (extended abstract). In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, June 1994. To appear.

[Hen88] F. Henglein. Type inference and semi-unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 184–197, July 1988.

[Hen92] F. Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, July 1992.

[HJ90] N. Heintze and J. Jaffar. A decision procedure for a class of Herbrand set constraints. In *Symposium on Logic in Computer Science*, pages 42–51, June 1990.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood Limited, 1987.

[HY88] P. Hudak and J. Young. A collecting interpretation of expressions (without powerdomains). In *Proceedings of the 15th Annual ACM Symposium on the Principles of Programming Languages*, pages 107–118, 1988.

[JM79] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[JM86] N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal fucntion graphs: Abridged version. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 296–306, January 1986.

[Mit91] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, July 1991.

[MPS84] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymophic types. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 165–174, January 1984.

[MW97] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the International Conference on Functional Programming*, June 1997.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, pages 269–281, April 1980. In LNCS No. 83.

[NN97] F. Nielson and H. R. Nielson. Infinitary control flow analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345, January 1997.

[Pal95] J. Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.

[PBJ$^+$91] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 67–78, January 1991.

[Pot96] F. Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 122–133, May 1996.

[PP97] A. Podelski and L. Pacholski. Set constraints—a pearl in research on constraints. In *Proceedings of the Third International Conference on the Principles and Practice of Constraint Programming*, October 1997. Springer-Verlag LNCS no. 1330.

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of the OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, November 1991.

[Pug91] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 4th Annual Conference on Supercomputing*, pages 4–13, November 1991.

[PW95] W. Pugh and D. Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):204–211, February 1995.

[R89́] D. Rémy. Type checking records and variants in a natural extension of ML. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 277–87, January 1989.

[Rey69] J. C. Reynolds. *Automatic Computation of Data Set Definitions*, pages 456–461. Information Processing 68. North-Holland, 1969.

[RM96] J. Rehof and T. A. Mogensen. Tractable constraints in finite semilattices. In *Proceedings of the 3rd International Static Analysis Symposium*, pages 285–295, September 1996.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, University of Copenhagen, October 1991.

[Shi88] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[TS96] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium*, volume 1145, pages 349–365. Springer Verlag, 1996.

[TT94] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

[Wad87] P. Wadler. Strictness analysis on non-flat domains (by Abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 266–275. Ellis Horwood Limited, 1987.

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.

[Wan93] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, pages 1–15, 1993.