Bounds Analysis by Abstract Interpretation*

Xiaolei Qian
Computer Science Laboratory
SRI International
qian@csl.sri.com

Allen Goldberg
Kestrel Institute
goldberg@kestrel.edu

Abstract

Abstract interpretation is a framework that precisely relates programs to their formal models. A model of a program retains its control structure but computes in a data domain that is less detailed, i.e. more abstract. Symbolically evaluating programs over the abstract domain yields analysis results that have been traditionally used by compilers for optimization.

This paper presents an abstract interpretation for a program language with set-theoretic data structures that automatically and efficiently obtains bounds and containment data that may be used to help select efficient representations for these abstract types. In this work an abstract context, i.e. model, consists of a deductively-closed collection of formulas in a simple logical theory. The technique is easily extensible to other high-level data abstractions, and readily incorporable into compilers for very-high-level languages.

1 Introduction

We develop a program analysis technique that derives the symbolic containment and membership assertions of set-theoretic data abstractions automatically and effectively. Abstract interpretation[3] is applied to construct a deductive semantics of programs. Concrete contexts are abstracted into sets of simple facts about program points called \textit{bound assertions}, and the abstract interpretation serves as a specialized theorem prover or \textit{fact gatherer}[11].

*Supported in part by the U.S. Air Force Rome Laboratory under contract F30602-86-C-0026.
which computes global bounds information. In this application of abstract interpretation, the abstract domain is a deductively-closed decidable theory, i.e., an abstract model of the program. For a syntactic construct $s$ and a bound assertion $p$, the abstract interpretation derives a post-condition of $p$ over $s[6]$. Based on the abstract interpretation, we show to perform bounds analysis that are of interest to data structure selection. The importance of inclusion and membership relationships to automatic data structure selection in very-high-level languages was first recognized in [12] and developed in [4, 5]. Techniques similar to data flow analysis were developed there for SETL, which start from an initial set of plausible relationships and iteratively eliminate those that might be false. Compared with the approach in [12], our approach does not require that all plausible inclusion and membership assertions be generated, and intra-function analysis is syntax-directed rather than iteratively approximated. Abstract interpretation was originally proposed in [2, 3] as a general framework for semantically correct program analysis. It has been applied to applicative languages to perform a variety of analysis tasks, such as strictness analysis in functional programs and mode analysis in logic programs[9]. An example of abstract interpretation applied to imperative languages is [1], where induction variables and recurrence relations in loops can be automatically recognized.

The paper is organized as follows. The syntax and semantics of our programming language are given in Sections 2 and 3. Section 4 defines abstract contexts and shows their correctness. Abstract interpretation is presented in Sections 5 and 6 together with their properties that are useful for bounds analysis. We discuss various types of bounds analysis in Section 7. Finally, Section 8 gives some concluding remarks.

2 Syntax

We define a simple programming language to illustrate the analysis method. The programming language has of three primitive datatypes: integer, symbol, and boolean; and five type constructors: pair, set, sequence, map, and (binary) relation. The grammar for the language is:

$$
\text{program}\n\text{fn} \equiv \text{function} \hspace{1em} \text{id}(\text{id}: \text{dl}, \ldots, \text{id}: \text{dl}) = \text{st} \hspace{1em} \text{return} \hspace{1em} \text{e} \\
\text{declaration}\n\text{dl} \equiv \text{integer} \mid \text{symbol} \mid \text{boolean} \\
\mid \text{pair}(\text{dl}, \text{dl}) \mid \text{set}(\text{dl}) \mid \text{sequence}(\text{dl}) \mid \text{map}(\text{dl}, \text{dl}) \mid \text{relation}(\text{dl}, \text{dl}) \\
\text{statement}$$
\begin{verbatim}
st \text{def} = id \leftarrow exp \\
| \text{id}(\exp) \leftarrow \exp \\
| \text{if} \exp \text{then} \; st \; \text{else} \; st \\
| (st; \ldots; st) \\
| \text{while} \exp \text{do} \; st \\
| \text{enumerate} \; \text{id} \; \text{over} \; \exp \; \text{do} \; st \\
| \text{let} \; (id = \exp, \ldots, id = \exp) \; st
\end{verbatim}

The expressions are formed by standard operators on the primitive types and the operators for composite types given in figure 1.

In this paper we restrict our attention to an intraprocedural analysis of a single function. In [10] the generalization of our analysis method to interprocedural analysis is given, as well as full proofs and a description of applications.

A program is represented by a directed graph \langle \text{Node}, \text{Arc} \rangle whose nodes correspond to syntactic constructs defined in the grammar, and whose arcs denote to control flow. Arcs are specified by predecessor and successor relationships, which are mappings over nodes. Every statement is represented by a (hyper) node with exactly one incoming arc and one outgoing arc, as shown in Figure 2. The incoming node and outgoing nodes of a function, are called the \text{entry} and \text{exit} nodes respectively. The unique arc from the entry node is called the \text{entry arc}, and the unique node to the exit node is called the \text{exit arc}.

For an if, test, or choice node \(n\), its two successor nodes are denoted by \text{succ}_i(n) and \text{succ}_f(n) respectively. For a test or choice node \(n\), its two predecessor nodes are denoted by \text{pred}_i(n) and \text{pred}_f(n) respectively. For an endif node, its two predecessor nodes are denoted by \text{pred}_i(n) and \text{pred}_f(n) respectively. For any other kind of node \(n\), \text{succ}(n) and \text{pred}(n) denote its successor and predecessor nodes respectively.

As an example, the program in Figure 3 computes a topological sort from a partial order \(R\) over a set of symbols \(S\). \text{Succ} is a map from an element of \(S\) to its direct successors in the partial order. A total order consistent with the partial order is accumulated in the sequence \(\text{result}\). \text{NumPred} is a map that maps an element of \(S\) to the number of elements of \(S\) not yet inserted into the total order \(\text{result}\). \text{MinEls} is the set of elements of \(S\) whose predecessors are elements of \(\text{result}\). The directed graph representation of the program is shown in Figure 4.
<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td>$x.1$</td>
<td>The first component of pair $x$</td>
</tr>
<tr>
<td></td>
<td>$x.2$</td>
<td>The second component of pair $x$</td>
</tr>
<tr>
<td></td>
<td>$(x, y)$</td>
<td>The pair consisting of $x$ and $y$</td>
</tr>
<tr>
<td>Set</td>
<td>$\emptyset$</td>
<td>Constant denoting the empty set</td>
</tr>
<tr>
<td></td>
<td>$\text{arb}(x)$</td>
<td>Select an arbitrary element of set $x$ (partial and nondeterministic)</td>
</tr>
<tr>
<td></td>
<td>$x$ with $y$</td>
<td>Add element $y$ to set $x$</td>
</tr>
<tr>
<td></td>
<td>$x$ less $y$</td>
<td>Remove element $y$ from set $x$</td>
</tr>
<tr>
<td></td>
<td>$\text{notempty}(x)$</td>
<td>Test if set $x$ is not empty</td>
</tr>
<tr>
<td></td>
<td>$x \in y$</td>
<td>Set membership</td>
</tr>
<tr>
<td></td>
<td>$x \subseteq y$</td>
<td>Set inclusion</td>
</tr>
<tr>
<td></td>
<td>$x \cup y$</td>
<td>Set union</td>
</tr>
<tr>
<td></td>
<td>$x \cap y$</td>
<td>Set intersection</td>
</tr>
<tr>
<td></td>
<td>$\text{power}(x)$</td>
<td>Powerset of set $x$</td>
</tr>
<tr>
<td></td>
<td>${ f(x) : x \in y \land p(x) }$</td>
<td>Set comprehension</td>
</tr>
<tr>
<td>Sequence</td>
<td>$[]$</td>
<td>Constant denoting the empty sequence</td>
</tr>
<tr>
<td></td>
<td>$\text{first}(x)$</td>
<td>The first element of sequence $x$ (partial)</td>
</tr>
<tr>
<td></td>
<td>$\text{append}(x, y)$</td>
<td>Append element $y$ to the end of sequence $x$</td>
</tr>
<tr>
<td></td>
<td>$\text{range}_{\text{seq}}(x)$</td>
<td>The set of elements in sequence $x$</td>
</tr>
<tr>
<td></td>
<td>$x \in_{\text{seq}} y$</td>
<td>Sequence membership</td>
</tr>
<tr>
<td></td>
<td>$[f(x) : x \in_{\text{seq}} y \land p(x)]$</td>
<td>Sequence comprehension</td>
</tr>
<tr>
<td>Map</td>
<td>$\emptyset$</td>
<td>Constant denoting the empty map</td>
</tr>
<tr>
<td></td>
<td>$x(y)$</td>
<td>Pointwise selection of map $x$ at point $y$ (partial)</td>
</tr>
<tr>
<td></td>
<td>$x \circ (y \rightarrow z)$</td>
<td>Pointwise assignment changing the value of map $x$ at point $y$ to $z$</td>
</tr>
<tr>
<td></td>
<td>$\text{domain}(x)$</td>
<td>The set of elements in the domain of map $x$</td>
</tr>
<tr>
<td></td>
<td>$\text{range}(x)$</td>
<td>The set of elements in the range of map $x$</td>
</tr>
<tr>
<td></td>
<td>${ f(x) \rightarrow g(x) : x \in y \land p(x) }$</td>
<td>Map comprehension</td>
</tr>
<tr>
<td>Relation</td>
<td>$\text{domain}_{\text{rel}}(x)$</td>
<td>Projection of binary relation $x$ onto its first component</td>
</tr>
<tr>
<td></td>
<td>$\text{range}_{\text{rel}}(x)$</td>
<td>Projection of binary relation $x$ onto its second component</td>
</tr>
</tbody>
</table>

Figure 1: Operators on Composite Data Types
Figure 2: Program Graphs
function TopSort(S: set(symbol), R: relation(symbol, symbol)) =

s0 let (result = [],
    Succ = \{ v → \{ \}: v ∈ S \},
    NumPred = \{ v → 0: v ∈ S \})

s1 (enumerate z over R do
    let (x = z.1, y = z.2)
        (Succ(x) ← Succ(x) with y;
         NumPred(y) ← NumPred(y) + 1;)
    s5 let (MinEls = { })

s6 (enumerate v over S do
    if NumPred(v) = 0 then MinEls ← MinEls with v;
    s8 while ¬(MinEls = { }) do
        let (a = arb(MinEls))
            (MinEls ← MinEls less a;
             result ← append(result, a);
             s11 foreach w in Succ(a) do
                 (NumPred(w) ← NumPred(w) - 1;
                  if NumPred(w) = 0 then MinEls ← MinEls with w))))

s15 return result

Figure 3: Topological Sorting Program

Figure 4: Graph Representation of Topological Sorting Program
3 Semantics

Following the Cousots [3], we define the semantic domain \textbf{Value} as the collection of values that program variables can take on; the semantic domain \textbf{Id} as the collection of program variables; the semantic domain \textbf{Exp} as the collection of expressions over the identifiers and operations; the semantic domain \textbf{Environment} as the mappings from \textbf{Id} to \textbf{Value}; and the semantic domain \textbf{State} as the collection of flow-graph arc and environment pairs, called states.

\textbf{Value} \; \overset{\text{def}}{=} \; \text{Integer} + \text{Boolean} + \text{Symbol} + \cdots

\textbf{Environment} \; \overset{\text{def}}{=} \; \text{Id} \rightarrow \text{Value}

\textbf{State} \; \overset{\text{def}}{=} \; \text{Arc} \times \text{Environment}

The semantic domain for representing the set-theoretic types are the hereditarily finite sets [7]. The meaning of an expression in an environment is defined by the evaluation function \(\text{val}: \text{Exp} \rightarrow (\text{Environment} \rightarrow \mathcal{P}(\text{Value}))\). Since there is a non-deterministic operator \(\text{arb}\), \(\text{val}\) maps an expression and an environment to a set of values. For expression \(e\) and environment \(r\), the meaning of \(e\) in \(r\) is denoted by \(\text{val}[e]r\). For most expressions, \(\text{val}\) is straightforward. For example

\[
\text{val}[S \text{ with } x]r = \{S_v \cup \{x_v\} | S_v \in \text{val}[S]r \land x_v \in \text{val}[x]r\}
\]

and

\[
\text{val}[\text{arb}S]r = \bigcup \text{val}[S]r
\]

The semantics of programs is defined using a state transition function \(\text{next}: \text{State} \rightarrow \mathcal{P}(\text{State})\), which maps a state to a collection of states. For arc \(\langle m, n \rangle\) and environment \(r\), \(\text{next}(\langle m, n \rangle, r)\) is defined as follows.

- If \(n\) is an assignment node representing the statement \(v \leftarrow e\), then the next state is in \(\{\langle n, \text{succ}(n) \rangle, r \circ (v \rightarrow x) \mid x \in \text{val}[e]r\}\). If \(n\) is an assignment node representing the statement \(v(d) \leftarrow e\), then the next state is in \(\{\langle n, \text{succ}(n) \rangle, r \circ (v \rightarrow \text{val}[v]r \circ (x \rightarrow y)) \mid x \in \text{val}[d]r \land y \in \text{val}[e]r\}\).

- If \(n\) is an if node representing the statement \(\text{if } q \text{ then } s_1 \text{ else } s_2\), then the collection of possible next states contains \(\langle n, \text{succ}(n) \rangle, r\) if \(\text{true} \in \text{val}[q]r\) and \(\langle n, \text{succ}(n) \rangle, r\) if \(\text{false} \in \text{val}[q]r\).

- If \(n\) is an endif node, then the next state is \(\langle n, \text{succ}(n) \rangle, r\).

- If \(n\) is a let node representing the statement \(\text{let } (v_1 = e_1, \ldots, v_l = e_l) s\), then the next state is in \(\{\langle n, \text{succ}(n) \rangle, r \circ (v_i \rightarrow x_i)_{i=1}^l \mid \land_{i=1}^l x_i \in \text{val}[e_i]r\}\).
• If \( n \) is an endlet node representing the statement \( \text{let} \ (v_1 = e_1, \ldots, v_l = e_l) \) \( s \), then the next state is \( \langle \langle n, \text{succ}(n) \rangle, \{v \rightarrow r[v] | v \in \text{domain}(r) \land v \notin \{v_1, \ldots, v_l\} \} \rangle \).

• If \( n \) is a test node representing the statement \( \text{while} \ q \ \text{do} \ s \), then the collection of possible next states contains \( \langle \langle n, \text{succ}(n) \rangle, r \rangle \) if \( \text{true} \in \text{val}[q]r \) and \( \langle \langle n, \text{succ}(n) \rangle, r \rangle \) if \( \text{false} \in \text{val}[q]r \).

• The semantics of the statement \( \text{enumerate} \ v \ \text{over} \ S \ \text{do} \ s \) is equivalent to that of the let statement

\[
\text{let} \ (S_{\text{new}} = S) \\
\quad \text{while} \ \neg (S_{\text{new}} = \{\}) \ \text{do} \\
\quad \quad \text{let} \ (v = \text{arb}(S_{\text{new}})) \\
\quad \quad \quad (s; \\
\quad \quad \quad \quad S_{\text{new}} \leftarrow S_{\text{new}} \ \text{less} \ v). 
\]

The static semantics of programs utilizes the notion of (concrete) contexts, which are collections of environments, i.e. \( \text{Context} \triangleq \mathcal{P}(\text{Environment}) \). Boolean expression \( e \) is true in concrete context \( c \) if \( \text{val}[e]r = \{\text{true} \} \) for every environment \( r \) in \( c \). Every arc is associated with a concrete context by the function \( \text{cont} : \text{Arc} \rightarrow \text{Context} \).

An input environment for a function \( h \) is a mapping from \( h \)'s formal parameters to values. The concrete context \( c_{\text{in}} \) associated with the (unique) input arc entry consists of a collection of input environments. The concrete context associated with an arc \( a \), \( \text{cont}(a) \), is defined as the collection of environments on \( a \) that can be obtained by applying the state transition function \( \text{next} \) a finite number of times to the input environment:

\[
\{r | (\exists i \geq 0)(\exists r \in c_{\text{in}})(\langle a, r \rangle \in \text{next}^i(\text{entry}, r))\}. 
\]

Finally, given an input environment \( r \), the concrete semantics of a function function declared as \( \text{function} \ h(v_1 : T_1, \ldots, v_l : T_l) = \text{st return} \ e \) is \( \{\text{val}[e]r' | r' \in \text{cont}(\text{exit})\} \).

4 Abstract Contexts

Abstract contexts are collections of atomic formulas over program variables of specific syntactic forms called bound assertions, that are closed under logical entailment.

4.1 Bound Assertions

The language of bound assertions is a restriction of the boolean expression available in the full programming language. There is one constant \( \{\} \). The bound terms, are formed from constants and variables using the following operations form the programming language:

\[
x, x(y), x.1, x.2, \text{range}_{\text{seq}}(x), \text{domain}(x), \text{range}(x), \text{domain}_{\text{rel}}(x), \text{range}_{\text{rel}}(x), \text{power}(x) 
\]
for terms $x$ and $y$ of appropriate types. The bound assertions are atomic formulas of the form
\[ \text{notempty}(t), t_1 = t_2, t_1 \in t_2, t_1 \subseteq t_2 \]
for bound terms $t, t_1$, and $t_2$ of appropriate types. We use $S_V$ to denote a collection of bound assertions whose variables are exactly those in $V$.

The $*$-closure of a collection of bound assertions $S_V$ over program variables $V$, denoted by $S^*_V$, is the collection of bound assertions over $V$ that are logically-entailed from $S_V$. $S_V$ is consistent if bound assertion notempty($\{\}$) is not in $S^*_V$. For a program whose program variables are contained in the set $V$ the abstract contexts are the sets $S^*_V$.

4.2 Correctness

Suppose that $g$ is a program with program variables $V$. An abstract interpretation of $g$ consists of a lattice of abstract contexts, ordered by set inclusion with bottom $\{\}$, and an interpretation function. It is correct if it approximates the static semantics or concrete interpretation of $g$. According to the Cousots [3], this amounts to the construction of two functions $\alpha$ and $\gamma$ such that $(\alpha, \gamma)$ forms an adjoined pair of functions: $\alpha \circ \gamma = \text{identity}$ and $\gamma \circ \alpha \supseteq \text{identity}$. The abstraction function $\alpha$ maps concrete contexts to abstract contexts. For every concrete context $c$, $\alpha(c)$ is the collection of bound assertions over $V$ that are true in every environment contained in $c$.

The concretization function $\gamma$ maps abstract contexts to concrete contexts. For every abstract context $c$, $\gamma(c)$ is the collection of environments in which every bound assertion in $c$ is true.

**Theorem 1** For abstract context $c$, $\alpha(\gamma(c)) = c$. For concrete context $c$, $c \subseteq \gamma(\alpha(c))$.

*Proof* Suppose that $c$ is an abstract context. For every bound assertion $p \in c$ and every environment $r \in \gamma(c)$, we have that $r \models p$. Hence $p \in \alpha(\gamma(c))$.

Let $p \in \alpha(\gamma(c))$. Then we have that for every model in which every formula of $c$ is true, $p$ is also true. Thus $c \models p$. But since an abstract context is closed under entailment, $p \in c$.

Suppose that $c$ is a concrete context. For every environment $r \in c$ and every bound assertion $p \in \alpha(c)$, we have that $r \models p$. Hence $r \in \gamma(\alpha(c))$. \hfill $\Box$

5 Abstraction Functions

To perform abstract interpretation, we need abstraction functions that map set-theoretic expressions and boolean expressions in our programming language to logically related bound terms and bound assertions respectively. These abstraction functions can be viewed as abstract interpretations of the corresponding expressions.
5.1 Canonicalization

The first step is to transform set-theoretic expressions and boolean expressions into logically equivalent canonicalized forms that are closer in syntax to bound terms and bound assertions. The canonicalization is achieved by applying the boolean simplification rules in, for example, [8] and the following transformation rules repeatedly until no rules are applicable:

\[
\begin{align*}
\text{range}_{\text{seq}}([]) & \iff \emptyset \\
\text{domain}([[]]) & \iff \emptyset \\
\text{range}([[]]) & \iff \emptyset \\
\text{notempty}([\{\}]) & \iff false \\
\text{notempty}(t_1 \text{ with } t_2) & \iff true \\
\text{notempty}(t_1 \cup t_2) & \iff \text{notempty}(t_1) \lor \text{notempty}(t_2) \\
\neg(t = \{\}) & \iff \text{notempty}(t) \\
t_1 \in t_2 & \iff t_1 \in \text{range}_{\text{seq}}(t_2) \\
\text{arb}(t_1) \in t_2 & \iff \text{notempty}(t_1) \land t_1 \subseteq t_2 \\
p[y](x) & \iff x \in \text{domain}(y) \land p(y(x)) \\
t_1 \in \text{power}(t_2) & \iff t_1 \subseteq t_2 \\
(t_1 \cup t_2) \subseteq t_3 & \iff (t_1 \subseteq t_3) \land (t_2 \subseteq t_3) \\
t_1 \subseteq (t_2 \cap t_3) & \iff (t_1 \subseteq t_2) \land (t_1 \subseteq t_3) \\
(t_1 \text{ with } t_2) \subseteq t_3 & \iff (t_1 \subseteq t_3) \land (t_2 \subseteq t_3) \\
t_1 \subseteq (t_2 \text{ less } t_3) & \iff (t_1 \subseteq t_2) \land \neg(t_1 = t_3) \\
\text{range}_{\text{seq}}(\text{append}(t_1, t_2)) & \iff \text{range}_{\text{seq}}(t_1) \text{ with } t_2 \\
\{v: v \in t\} & \iff t & t \text{ has type } set(\delta) \\
\{v: v \in t\} & \iff \text{range}_{\text{seq}}(t) \\
[v: v \in t] & \iff t & t \text{ has type } seq(\delta) \\
\text{domain}(m \circ (v \to e)) & \iff \text{domain}(m) \text{ with } v \\
\text{range}(m \circ (v \to e)) & \iff (\text{range}(m) \text{ less } m(v)) \text{ with } e \\
\text{domain}([d \to r: d \in t]) & \iff t & t \text{ has type } set(\delta) \\
\text{domain}([d \to r: d \in t]) & \iff \text{range}_{\text{seq}}(t) & t \text{ has type } seq(\delta) \\
t_1 = t_2 & \iff t_1 \subseteq t_2 \land t_2 \subseteq t_1 & t_1, t_2 \text{ are sets}
\end{align*}
\]

Most transformation rules are straightforward except the ones that involve partial operators arbitrary selection, and pointwise selection. The rules generate the necessary applicability assertions, and preserve logical equivalence for defined terms.

5.2 Closure

Abstract context are collections of bound assertions closed under logical entailment. To generate closed sets we introduce a deductive system and show it complete. The +-closure of
a collection of bound assertions $S_V$ over program variables $V$, denoted by $S_V^+$, is the collection of bound assertions over $V$ that are inferred from $S_V$ by following rules of inference:

**Tautology**

\[
\{\} \subseteq x \\
x \subseteq x
\]

**Empty Set**

\[
\begin{align*}
x \in y & \Rightarrow \text{notempty}(y) \\
\text{notempty}(x), x \subseteq y & \Rightarrow \text{notempty}(y)
\end{align*}
\]

**Subset**

\[
\begin{align*}
x \subseteq y, y \subseteq x & \Rightarrow x = y \\
x = y & \Rightarrow x \subseteq y, y \subseteq x \\
x \subseteq y, y \subseteq z & \Rightarrow x \subseteq z
\end{align*}
\]

**Power Set**

\[
\begin{align*}
x \subseteq y & \Rightarrow \text{power}(x) \subseteq \text{power}(y) \\
\text{power}(x) \subseteq \text{power}(y) & \Rightarrow x \subseteq y \\
x \in \text{power}(y) & \Rightarrow x \subseteq y
\end{align*}
\]

**Binary Relation**

\[
\begin{align*}
x \in y & \Rightarrow x.1 \in \text{domain}_{rel}(y) \\
x \in y & \Rightarrow x.2 \in \text{range}_{rel}(y)
\end{align*}
\]

**Map Range**

\[
\begin{align*}
x \in \text{domain}(y) & \Rightarrow y(x) \in \text{range}(y) \\
y(x) \in \text{range}(y) & \Rightarrow x \in \text{domain}(y)
\end{align*}
\]

**Equality**

\[
\begin{align*}
x = y, y = z & \Rightarrow x = z \\
x = y & \Rightarrow f(x) = f(y)
\end{align*}
\]

These rules are a specialized theorem proving system for bound assertions. For any given finite collection of program variables, by restricting the use of the axiom

\[
x \subseteq y \\
\text{power}(x) \subseteq \text{power}(y)
\]
in the obvious way, only a finite number of bound terms and hence bound assertions can be formed. Therefore, inference in this system always terminates. It is also easy to verify that our axiom system is sound. Theorem 2 shows that it is complete with respect to bound assertions.

**Theorem 2** If $S_V$ is consistent, then $S_V^+ = S_V^\ast$.

**Proof** (proof outline.) Suppose that there is a bound assertion $p \in S_V^+$ but $p \notin S_V^\ast$. We construct a model of $S_V^+$ in which $p$ is false. This is a contradiction because $(S_V^+)^\ast = S_V^\ast$. To construct the model we build a graph whose nodes are bound terms and whose edges denote
membership or subset relations, insuring that any relationship that is provable is explicitly represented.

5.3 Bound Term Abstraction

We define two abstraction functions that map set-theoretic expressions to logically-related bound terms. $A_b(t)$ maps a set-valued expression $t$ to a collection of bound terms that are subsets of $t$, while $A_p(t)$ maps a set-valued expression $t$ to a collection of bound terms that are supersets of $t$. In other words, for every $t_1 \in A_b(t)$ and $t_2 \in A_p(t)$, $t_1 \subseteq t$ and $t \subseteq t_2$ are valid. If $t$ is a bound term, then $A_b(t) = A_p(t) = \{t\}$. If $t$ is not a bound term and is not in one of the forms covered by the rules below, then the abstraction functions return empty. Otherwise they are computed by the following rules, where $A_w$ and $A_s$ are the bound-assertion abstraction functions to be defined in the next section:

\[
\begin{align*}
A_b(t_1 \text{ with } t_2) & \overset{\text{def}}{=} A_b(t_1) \\
A_b(t_1 \cup t_2) & \overset{\text{def}}{=} A_b(t_1) \cup A_b(t_2) \\
A_b(t_1 \cap t_2) & \overset{\text{def}}{=} A_b(t_1) \cap A_b(t_2) \\
A_b(\{v : p\}) & \overset{\text{def}}{=} \{ x | v \in x \in A_s(p) \} \\
A_b(\text{range}_{seq}(\{v : p\})) & \overset{\text{def}}{=} \{ x | v \in x \in A_s(p) \} \\
A_b(\text{domain}(\{d \rightarrow r : p\})) & \overset{\text{def}}{=} \{ x | d \in x \in A_s(p) \} \\
A_b(\text{range}(\{d \rightarrow r : p\})) & \overset{\text{def}}{=} \{ x | r \in x \in A_s(p) \}
\end{align*}
\]

\[
\begin{align*}
A_p(t_1 \text{ less } t_2) & \overset{\text{def}}{=} A_p(t_1) \\
A_p(t_1 - t_2) & \overset{\text{def}}{=} A_p(t_1) \\
A_p(t_1 \cup t_2) & \overset{\text{def}}{=} A_p(t_1) \cup A_p(t_2) \\
A_p(t_1 \cap t_2) & \overset{\text{def}}{=} A_p(t_1) \cap A_p(t_2) \\
A_p(\{v : p\}) & \overset{\text{def}}{=} \{ x | v \in x \in A_w^+(p) \} \\
A_p(\text{range}_{seq}(\{v : p\})) & \overset{\text{def}}{=} \{ x | v \in x \in A_w^+(p) \} \\
A_p(\text{domain}(\{d \rightarrow r : p\})) & \overset{\text{def}}{=} \{ x | d \in x \in A_w^+(p) \} \\
A_p(\text{range}(\{d \rightarrow r : p\})) & \overset{\text{def}}{=} \{ x | r \in x \in A_w^+(p) \}
\end{align*}
\]

5.4 Bound Assertion Abstraction

We define two abstraction functions that map boolean expressions to logically related bound assertions. $A_w(p)$ maps a boolean expression $p$ to a collection of bound assertions that are logically weaker than $p$. In other words, for every $q \in A_w(p)$, $p \rightarrow q$ is valid. Likewise, $A_s(p)$ maps a boolean expression $p$ to a collection of collections of bound assertions that are
logically stronger than $p$. In other words, for every $S \in \mathcal{A}_s(p)$, $\bigwedge_{q \in S} q \rightarrow p$ is valid. If $p$ is a bound assertion, then $\mathcal{A}_w(p) = \{p\}$ and $\mathcal{A}_w(p) = \{\{p\}\}$. If $p$ is not a bound assertion and is not in one of the forms covered by the rules below, then the abstraction functions return empty. Otherwise they are computed by the following rules, where $\mathcal{A}_b$ and $\mathcal{A}_p$ are the bound-term abstraction functions defined in the previous section:

\[
\begin{align*}
\mathcal{A}_w(p \land q) & \overset{\text{def}}{=} \mathcal{A}_w(p) \cup \mathcal{A}_w(q) \\
\mathcal{A}_w(p \lor q) & \overset{\text{def}}{=} \mathcal{A}_w(p) \cap \mathcal{A}_w(q) \\
\mathcal{A}_w(\text{notempty}(t)) & \overset{\text{def}}{=} \{\text{notempty}(x)| x \in \mathcal{A}_p(t)\} \\
\mathcal{A}_w(x_1 \in t_2) & \overset{\text{def}}{=} \{x_1 \in x_2|x_2 \in \mathcal{A}_p(t_2)\} \quad x_1 \text{ is a bound term} \\
\mathcal{A}_w(t_1 \subseteq t_2) & \overset{\text{def}}{=} \{x_1 \subseteq x_2|x_1 \in \mathcal{A}_b(t_1) \land x_2 \in \mathcal{A}_p(t_2)\} \\
\mathcal{A}_w(t_1 = \text{arb}(t_2)) & \overset{\text{def}}{=} \mathcal{A}_w(\text{notempty}(t_2) \land t_1 \in t_2) \\
\mathcal{A}_w(t_1 = \text{first}(t_2)) & \overset{\text{def}}{=} \mathcal{A}_w(\text{notempty}(\text{range}_{\text{seq}}(t_2)) \land t_1 \in \text{range}_{\text{seq}}(t_2)) \\
\mathcal{A}_w(t_1 = t_2(t_3)) & \overset{\text{def}}{=} \mathcal{A}_w(t_3 \in \text{domain}(t_2) \land t_1 \in \text{range}(t_2)) \quad t_2 \text{ has type map}(\delta, \sigma) \\
\mathcal{A}_w(t_1 = t_2) & \overset{\text{def}}{=} \mathcal{A}_w(\text{range}_{\text{seq}}(t_1) \subseteq \text{range}_{\text{seq}}(t_2)) \\
& \quad \cup \mathcal{A}_w(\text{domain}(t_2) \subseteq \text{domain}(t_1)) \\
& \quad \cup \mathcal{A}_w(\text{range}(t_1) \subseteq \text{range}(t_2)) \\
& \quad \cup \mathcal{A}_w(\text{range}(t_2) \subseteq \text{range}(t_1)) \quad \text{sequence equality} \\
\mathcal{A}_w(t_1 = t_2) & \overset{\text{def}}{=} \mathcal{A}_w(\text{domain}(t_1) \subseteq \text{domain}(t_2)) \\
& \quad \cup \mathcal{A}_w(\text{range}(t_2) \subseteq \text{range}(t_1)) \quad \text{map equality}
\end{align*}
\]

\[
\begin{align*}
\mathcal{A}_s(p \land q) & \overset{\text{def}}{=} \{x_1 \cup x_2|x_1 \in \mathcal{A}_s(p) \land x_2 \in \mathcal{A}_s(q)\} \\
\mathcal{A}_s(p \lor q) & \overset{\text{def}}{=} \mathcal{A}_s(p) \cup \mathcal{A}_s(q) \\
\mathcal{A}_s(\text{notempty}(t)) & \overset{\text{def}}{=} \{\text{notempty}(x)| x \in \mathcal{A}_b(t)\} \\
\mathcal{A}_s(x_1 \in t_2) & \overset{\text{def}}{=} \{x_1 \in x_2|x_2 \in \mathcal{A}_b(t_2)\} \quad x_1 \text{ is a bound term} \\
\mathcal{A}_s(t_1 \subseteq t_2) & \overset{\text{def}}{=} \{x_1 \subseteq x_2|x_1 \in \mathcal{A}_b(t_1) \land x_2 \in \mathcal{A}_b(t_2)\}
\end{align*}
\]

5.5 Example

As an example from the topsort program of Figure 3, $\mathcal{A}_w$ applied to the canonicalized boolean expression $\text{Succ} = \{v \rightarrow \{\}: v \in S\}$ is the union of $\mathcal{A}_w$ applied to the four boolean expressions below (after canonicalization):

\[
\begin{align*}
\text{domain}(\text{Succ}) & \subseteq S \\
S & \subseteq \text{domain}(\text{Succ}) \\
\text{range}(\text{Succ}) & \subseteq \text{range}(\{v \rightarrow \{\}: v \in S\}) \\
\text{range}(\{v \rightarrow \{\}: v \in S\}) & \subseteq \text{range}(\text{Succ})
\end{align*}
\]

The first two expressions are already in the form of bound assertions. $\mathcal{A}_w$ applied to the third expression results in the bound assertion $\text{range}(\text{Succ}) \subseteq \text{power}(S)$, because bound assertion
For bound term \( t \in \mathcal{A}_b(t_1), t_2 \subseteq t_1 \) is valid. For every bound term \( t_2 \in \mathcal{A}_p(t_1), t_1 \subseteq t_2 \) is valid. For every bound assertion \( q \in \mathcal{A}_w(p), p \rightarrow q \) is valid. For every collection of bound assertions \( S \in \mathcal{A}_s(p), \land_{q \in S} q \rightarrow p \) is valid.

Proof We prove by induction on the structure of set-theoretic and boolean expressions. For bound term \( t \in \mathcal{A}_b(t_1 \cup t_2) \), we have that either \( t \in \mathcal{A}_b(t_1) \) or \( t \in \mathcal{A}_b(t_2) \) by definition. But \( t \subseteq t_1 \) or \( t \subseteq t_2 \) is valid according to induction hypothesis on \( \mathcal{A}_b \). Hence \( t \subseteq t_1 \cup t_2 \) is valid. For bound term \( t \in \mathcal{A}_b(\{v:p\}) \), we have that \( \{v \in t\} \in \mathcal{A}_s(p) \) by definition, and \( v \in t \rightarrow p \) is valid by induction hypothesis on \( \mathcal{A}_s \). So \( \{v:v \in t\} \subseteq \{v:p\} \) and hence \( t \subseteq \{v:p\} \) are valid. Other cases for \( \mathcal{A}_b \) and \( \mathcal{A}_p \) are similar.

For bound assertion \( q \in \mathcal{A}_w(t_1 \subseteq t_2), q \) must be of the form \( x_1 \subseteq x_2 \) by definition. Furthermore, \( x_1 \subseteq t_1 \) and \( t_2 \subseteq x_2 \) are valid by induction hypothesis on \( \mathcal{A}_b \) and \( \mathcal{A}_w \). It follows that \( t_1 \subseteq t_2 \rightarrow q \) is valid. For \( q \in \mathcal{A}_w(t_1 = \text{arb}(t_2)), q \in \mathcal{A}_w(\text{notempty}(t_2)) \) or \( q \in \mathcal{A}_w(t_1 \subseteq t_2) \) by definition. By induction hypothesis on \( \mathcal{A}_w \), \( \text{notempty}(t_2) \rightarrow q \) or \( t_1 \subseteq t_2 \rightarrow q \) is valid. In other words, \( \text{notempty}(t_2) \land t_1 \subseteq t_2 \rightarrow q \) is valid. Hence \( t_1 = \text{arb}(t_2) \rightarrow q \) is valid since \( t_1 = \text{arb}(t_2) \rightarrow \text{notempty}(t_2) \land t_1 \subseteq t_2 \rightarrow q \) is valid. Other cases for \( \mathcal{A}_w \) are similar.

For collection of bound assertions \( S \in \mathcal{A}_s(p \land q) \), there exists \( S_1 \) and \( S_2 \) such that \( S = S_1 \cup S_2, S_1 \in \mathcal{A}_s(p) \), and \( S_2 \in \mathcal{A}_s(q) \). By induction hypothesis on \( \mathcal{A}_s \), \( \land_{r \in S_1} r \rightarrow p \) and \( \land_{r \in S_2} r \rightarrow q \) are valid. Hence \( \land_{r \in S} r \rightarrow (p \land q) \) is valid. Other cases for \( \mathcal{A}_s \) are similar. \( \square \)

## 6 Interpretation Function

The abstract interpretation \( \mathfrak{I} \) maps an abstract context \( c \) and a arc \( \langle m, n \rangle \) to another abstract context \( c' \). Intuitively, if \( c \) is true before executing \( m \), then \( c' \) is true after executing \( m \) on the arc \( \langle m, n \rangle \). In other words, \( \mathfrak{I} \) computes post-conditions of bound assertions over syntactic constructs. If \( c' = \emptyset \), then the post-condition is \( \text{true} \). If node \( n \) has only one outgoing arc, then \( \mathfrak{I}(c, n) \) stands for \( \mathfrak{I}(c, \langle n, \text{succ}(n) \rangle) \). If node \( n \) represents statement \( s \), then \( \mathfrak{I}(c, s) \) stands for \( \mathfrak{I}(c, n) \). For notational convenience, \( \mathfrak{I}(S, n) \) denotes \( \mathfrak{I}(S^+, n) \) for set of bound assertions \( S \), and \( \mathfrak{I}(p, n) \) denotes \( \mathfrak{I}(\{p\}, n) \) for bound assertion \( p \). For expression \( e \), \( e[e'/v] \) denotes the expression identical to \( e \) except that occurrences of \( v \) in \( e \) are replaced by \( e' \).

### 6.1 Non-Looping Constructs

The abstract interpretation of non-looping constructs is specified by the rules below. The interpretation of those syntactic constructs not listed is taken to be the identical mapping.
In the third and fourth rules, \( n \) denotes an if node with boolean expression \( q \).

\[
\begin{align*}
\mathcal{Z}(c, v \leftarrow e) & \overset{\text{def}}{=} (S \cup \{p | p \in c \land (\exists \delta')(c' \subseteq c \land c' \in \mathcal{A}_s(p[e/v]))\})^+ \\
& \quad \text{if } v \text{ occurs in } e \\
& \quad S = \{\} \text{ otherwise} \\
\mathcal{Z}(c, v(d) \leftarrow e) & \overset{\text{def}}{=} (S \cup \{p | p \in c \land (\exists \delta')(c' \subseteq c \land c' \in \mathcal{A}_s(p[v \circ (d \rightarrow e)/v]))\})^+ \\
& \quad \text{if } v \text{ occurs in } d \text{ or } e \\
& \quad S = \mathcal{A}_w(v = e) \text{ otherwise} \\
\mathcal{Z}(c, \langle n, \text{succ}_{\ell}(n) \rangle) & \overset{\text{def}}{=} (c \cup \mathcal{A}_w(q))^+ \\
\mathcal{Z}(c, \langle n, \text{succ}_{f}(n) \rangle) & \overset{\text{def}}{=} (c \cup \mathcal{A}_w(\neg q))^+ \\
\mathcal{Z}(c, \text{let } v_1 = e_1, \ldots, v_l = e_l) & \overset{\text{def}}{=} (c \cup \mathcal{A}_w(v_1 = e_1 \land \cdots \land v_l = e_l))^+ \\
\mathcal{Z}(c, \text{if } q \text{ then } s_1 \text{ else } s_2) & \overset{\text{def}}{=} \mathcal{Z}((c \cup \mathcal{A}_w(q))^+, s_1) \cap \mathcal{Z}((c \cup \mathcal{A}_w(\neg q))^+, s_2) \\
\mathcal{Z}(c, (s_1; \ldots; s_l)) & \overset{\text{def}}{=} \mathcal{Z}((c, s_1), (s_2; \ldots; s_l)) \\
\mathcal{Z}(c, \text{let } (v_1 = e_1, \ldots, v_l = e_l) s) & \overset{\text{def}}{=} \{p | p \in \mathcal{Z}((c \cup \mathcal{A}_w(v_1 = e_1 \land \cdots \land v_l = e_l))^+, s) \\
& \quad \land p \text{ is free of } v_1, \ldots, v_l \}
\end{align*}
\]

The interpretation for assignment statements needs further explanation. According to Hoare Logic [6], \( p \) is true after \( v \leftarrow e \) if \( p[e/v] \) is true before the assignment. Assuming that \( p \) is true before the assignment, then it is true after the assignment if \( p \rightarrow p[e/v] \) is true before the assignment. Given an abstract context \( c \) that is true before \( v \leftarrow e \) and \( p \in c \), \( p \) is true after the assignment if there exists \( c' \subseteq c \) such that \( c' \in \mathcal{A}_s(p[e/v]) \).

### 6.2 Looping Constructs

The basic idea of the rules for looping constructs is as follows. The only way that a bound assertion could be propagated through a looping construct is when it is an invariant of the loop. Therefore the rules propagate exactly those bound assertions true on entry to the loop that are verifiably loop invariants.

To verify that an abstract context \( c \) is an invariant of a loop with loop body \( s \), we need to show that if \( c \) is true before \( s \) then it is true after \( s \). By the definition of abstract interpretation, it is sufficient to show that \( \mathcal{Z}(c, s) \) logically implies \( c \).

An invariant of while statement \textbf{while } \( q \) \textbf{do } \( s \) is an abstract context \( \text{Inv} \) such that \( \text{Inv} \subseteq \mathcal{Z}((\text{Inv} \cup \mathcal{A}_w(q))^+, s) \). Invariants of enumerate statements can be defined similarly.

The rules for abstract interpretation can be stated as follows. Let \( n \) be the test node of while statement \textbf{while } \( q \) \textbf{do } \( s \). For abstract context \( c \), let \( \text{Inv} \) be the largest invariant of the statement such that \( \text{Inv} \subseteq c \):

\[
\begin{align*}
\mathcal{Z}(c, \langle n, \text{succ}_{\ell}(n) \rangle) & \overset{\text{def}}{=} (\text{Inv} \cup \mathcal{A}_w(q))^+ \\
\mathcal{Z}(c, \text{while } q \text{ do } s) & \overset{\text{def}}{=} (\text{Inv} \cup \mathcal{A}_w(\neg q))^+
\end{align*}
\]
The first rule for enumerate statements of the form `enumerate v over S do s` is the same if we replace `q` by `v ∈ S`. The second rule is specified as follows, since `v` should not be visible outside the statement:

\[ \exists(c, enumerate v over S do s) \overset{\text{def}}{=} Inv \]

### 6.3 Example

For the topsort program of Figure 3, we compute the interpretation of a collection `P` of bound assertions:

\[
\begin{align*}
p_1 & : \text{domain}(\text{ Succ }) = S \\
p_2 & : \text{range}(\text{ Succ }) \subseteq \text{power}(S) \\
p_3 & : \text{domain}_{rel}(R) \subseteq S \\
p_4 & : \text{range}_{rel}(R) \subseteq S
\end{align*}
\]

through the first enumerate statement `s_1`:

\[
\begin{align*}
s_1 : & \text{enumerate } z \text{ over } R \text{ do} \\
s_2 : & \text{let } (x = z.1, y = z.2) \\
s_3 : & (\text{ Succ } (x) \leftarrow \text{ Succ } (x) \text{ with } y; \\
s_4 : & \text{NumPred}(y) \leftarrow \text{NumPred}(y) + 1;
\end{align*}
\]

In addition to the bound assertions in `P`, `P' = (P \cup A_s(z \in R)^+` contains bound assertions `z.1 \in S, z.2 \in S, \text{ Succ } (z.1) \subseteq S`, etc. Assuming that `n` is the let node of `s_2`, we have that `\exists(P', n) = (P' \cup \{x = z.1, y = z.2\})^+`. Notice that

\[
\begin{align*}
A_s(p_1[\text{ Succ } \circ (x \to \text{ Succ } (x) \text{ with } y)/\text{ Succ }]) \\
&= A_s(\text{domain}(\text{ Succ }) \text{ with } x \subseteq S \land S \subseteq \text{domain}(\text{ Succ }) \text{ with } x) \\
&= \{\{\text{domain}(\text{ Succ }) \subseteq S, x \in S\} \cup \{S \subseteq \text{domain}(\text{ Succ })\}\} \\
&= \{p_1, x \in S\}
\end{align*}
\]

\[
\begin{align*}
A_s(p_2[\text{ Succ } \circ (x \to \text{ Succ } (x) \text{ with } y)/\text{ Succ }]) \\
&= A_s(\text{range}(\text{ Succ }) \text{ less } \text{ Succ } (x) \subseteq \text{power}(S) \land \text{ Succ } (x) \subseteq S \land y \in S) \\
&= \{\{\text{range}(\text{ Succ }) \subseteq \text{power}(S)\} \cup \{\text{ Succ } (x) \subseteq S, y \in S\}\} \\
&= \{p_2, \text{ Succ } (x) \subseteq S, y \in S\}
\end{align*}
\]

both of which contain subcollections of `\exists(P', n)`. Furthermore, `p_3, p_4` are not affected by `s_3`. Hence `P` can be propagated through `s_3` and therefore `s_2`. In other words, `P \subseteq \exists(P', s_2)`. In conclusion, all bound assertions in `P` are loop invariants of the enumerate statement `s_1` and can be propagated through it: `\exists(P, s_1) = P`. 
6.4 Correctness

Lemma 4  For abstract context $c$ and arc $a = \langle m, n \rangle$, $\exists(c, a)$ is an abstract context.

Proof  (proof outline.) $\exists$ has been defined so that it either explicitly closes the set using the inference procedure, or it follows inductively from the fact that $c$ is. □

Theorem 5  For abstract context $c$ and arc $a = \langle m, n \rangle$, if $c$ is true before executing $m$, then abstract context $\exists(c, a)$ is true after executing $m$ on arc $a$.

Proof  We show by induction on the structure of $m$. Most rules for non-looping constructs are self-explanatory. For the first assignment rule and every $p$ in $\exists(c, v \leftarrow e)$, if $p \in A_w(v = e)$ then $p$ is true after the assignment because $v = e$ is. Otherwise there exists $c' \subseteq c$ true before the assignment such that $c' \in A_i(p[e/v])$. Thus $p[e/v]$ is true before the assignment according to Theorem 3, and hence $p$ is true after the assignment.

For looping constructs, we show the case with while statement $s$ of the form while $q$ do $s'$. The case with enumerate statements is similar. Suppose that $c$ is true before executing $s$, and $Inv$ is the largest invariant of $s$ such that $Inv \subseteq c$. If $Inv$ and $q$ are true before executing $s'$, then $\exists((Inv \cup A_w(q))^+, s')$ is true after executing $s'$ by induction hypothesis, and hence $Inv$ is true after executing $s'$ since $Inv \subseteq \exists((Inv \cup A_w(q))^+, s')$. Therefore $\exists(c, s) = Inv$ is true after executing $s$ since $c$ is true before executing $s$ and $Inv \subseteq c$. □

7 Bounds Analysis

Bounds Analysis is the process of determining bound assertions over program variables that are true throughout the lifetime of the program variables involved, i.e., in all environments in which these program variables are defined. We demonstrate the process of bounds analysis using our topsort program. First we propagate bound assertions to the abstract contexts associated with all program arcs. Assuming that the input abstract context consists of two bound assertions:

\begin{align*}
\text{domain}_{rel}(R) &\subseteq S \\
\text{range}_{rel}(R) &\subseteq S
\end{align*}

After propagating through the let node of $s_0$, the abstract context contains in addition the following bound assertions:

\begin{align*}
\text{range}_{seq}(\text{result}) &\subseteq S \\
\text{domain}(\text{Succ}) & = S \\
\text{range}(\text{Succ}) &\subseteq \text{power}(S) \\
\text{domain}(\text{NumPred}) & = S
\end{align*}
The above collection of bound assertions can be propagated to the abstract context at the outgoing arc of the first statement of \( s_1 \), as demonstrated in Section 6.3. After propagating them through the let node of \( s_5 \), a new bound assertion \( \text{MinEls} \subseteq S \) is obtained. Let's denote these bound assertions by \( P \). Similar to Section 6.3 we can push \( P \) through the first statement of \( s_6 \). Notice that \( A_w(\neg(\text{MinEls} = \{\}) = \{\text{notempty}(\text{MinEls})\} \). In order to push \( P \) through \( s_8 \), we have to compute \( \exists((P \cup \{\text{notempty}(\text{MinEls})\})^+, s_9) \). After pushing \( P \cup \{\text{notempty}(\text{MinEls})\} \) through the let node of \( s_9 \), we get new bound assertions \( a \in \text{MinEls}, a \in S \), etc. Only \( P' = P \cup \{a \in S\} \) can be pushed through the first statement of \( s_{10} \), which can then be pushed through \( s_{11} \).

Notice that \( A_w(w \in \text{Succ}(a)) = \{a \in \text{domain}(\text{Succ}), w \in \text{Succ}(a)\} \), and \( w \in S \) is in the closure of \( P' \cup \{a \in \text{domain}(\text{Succ}), w \in \text{Succ}(a)\} \). Similar to Section 6.4, we can push \( P' \) through \( s_{13} \), hence \( \exists(P', s_{12}) = P' \). Finally, \( P \) can be pushed through \( s_9 \), therefore \( \exists(P, s_9) = P \). All bound assertions in \( P \) not mentioning \( \text{MinEls} \) reach the outgoing arc of \( s_5 \), and hence the outgoing arc of the function. The abstract contexts associated with other arcs can be computed in the similar way.

From the abstract interpretation, we conclude that \( S \) is a containment bound for \( \text{MinEls} \), a domain bound and an access bound for \( \text{Succ} \) and \( \text{NumPred} \), and a range bound for \( \text{result} \). In addition, \( \text{power}(S) \) is a range bound for \( \text{Succ} \).

An efficient implementation of tosort requires that access to the maps \( \text{NumPred} \) and \( \text{Succ} \) take constant time. This can be achieved by a data structure in which each element of \( S \) is a heap-allocated record containing fields that hold the corresponding range values of \( \text{NumPred} \), and \( \text{Succ} \). Elements of \( S \) appearing in \( \text{MinEls} \), range(\( \text{Succ} \)), \( R \), and \( \text{result} \) are represented by a pointers to the record. This representation requires the assertions developed by our analysis.

8 Conclusion

Bounds analysis, which is the process of determining symbolic bounds on program variables that are true throughout their lifetime, is central to the successful compilation of programs in very-high-level languages into efficient low-level implementation. We developed a technique for the automatic analysis of bounds information for several important set-theoretic data abstractions. Abstract interpretation is applied to associate a collection of bound assertions with every program point, based on which a variety of bounds analysis can be performed that are of interest to data structure selection. The abstract interpretation is kept computationally tractable by restricting ourselves to reasoning with bound assertions. Compared with the approach in [12], our approach does not require that all plausible inclusion and membership assertions be generated, and intra-function analysis is syntax-directed rather than iteratively approximated. The technique is easily extensible to other high-level data abstractions, and readily incorporable into compilers for very-high-level languages.
Acknowledgment

We thank Lee Blaine for our collaboration on the DTRE program transformation system, which has been a source of inspiration for the ideas presented here.

References


