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 *bound assertions*, and the abstract interpretation serves as a specialized theorem prover or *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:

program

$\mathbf{fn} \stackrel{\text{def}}{=} \mathbf{function}\ id(id\colon dl, \ldots, id\colon dl) = st\ \ \mathbf{return}\ \ e$

declaration

$\mathbf{dl} \stackrel{\text{def}}{=}$ integer | symbol | boolean
| pair(dl, dl) | set(dl) | sequence(dl) | map(dl, dl) | relation(dl, dl)

statement

$$
\begin{aligned}
\mathbf{st} \stackrel{\text{def}}{=} & \; id \leftarrow exp \\
& \mid id(exp) \leftarrow exp \\
& \mid \textbf{if } exp \textbf{ then } st \textbf{ else } st \\
& \mid (st; \ldots; st) \\
& \mid \textbf{while } exp \textbf{ do } st \\
& \mid \textbf{enumerate } id \textbf{ over } exp \textbf{ do } st \\
& \mid \textbf{let } (id = exp, \ldots, id = exp) \; st
\end{aligned}
$$

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 out 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 $\langle$ **Node** , **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 *entry* and *exit* nodes respectively. The unique arc from the entry node is called the *entry arc*, and the unique node to the exit node the is called the *exit arc*.

For an if, test, or choice node n, its two successor nodes are denoted by $succ_t(n)$ and $succ_f(n)$ respectively. For a test or choice node n, its two predecessor nodes are denoted by $pred_i(n)$ and $pred_l(n)$ respectively. For an endif node, its two predecessor nodes are denoted by $pred_t(n)$ and $pred_f(n)$ respectively. For any other kind of node n, $succ(n)$ and $pred(n)$ 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. *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 *result*. *NumPred* is a map that maps an element of S to the number of elements of S not yet inserted into the total order *result*. *MinEls* is the set of elements of S whose predecessors are elements of *result*. The directed graph representation of the program is shown in Figure 4.

Type	Expression	Remark
Pair	$x.1$	The first component of pair x
	$x.2$	The second component of pair x
	$\langle x, y \rangle$	The pair consisting of x and y
Set	$\{\,\}$	Constant denoting the empty set
	$\mathrm{arb}(x)$	Select an arbitrary element of set x (partial and nondeterministic)
	x with y	Add element y to set x
	x less y	Remove element y from set x
	$\mathrm{notempty}(x)$	Test if set x is not empty
	$x \in y$	Set membership
	$x \subseteq y$	Set inclusion
	$x \cup y$	Set union
	$x \cap y$	Set intersection
	$\mathrm{power}(x)$	Powerset of set x
	$\{f(x) \colon x \in y \wedge p(x)\}$	Set comprehension
Sequence	$[\,]$	Constant denoting the empty sequence
	$\mathrm{first}(x)$	The first element of sequence x (partial)
	$\mathrm{append}(x, y)$	Append element y to the end of sequence x
	$\mathrm{range}_{seq}(x)$	The set of elements in sequence x
	$x \in_{seq} y$	Sequence membership
	$[f(x) \colon x \in_{seq} y \wedge p(x)]$	Sequence comprehension
Map	$\{\mid\,\mid\}$	Constant denoting the empty map
	$x(y)$	Pointwise selection of map x at point y (partial)
	$x \circ (y \to z)$	Pointwise assignment changing the value of map x at point y to z
	$\mathrm{domain}(x)$	The set of elements in the domain of map x
	$\mathrm{range}(x)$	The set of elements in the range of map x
	$\{\mid f(x) \to g(x) \colon x \in y \wedge p(x) \mid\}$	Map comprehension
Relation	$\mathrm{domain}_{rel}(x)$	Projection of binary relation x onto its first component
	$\mathrm{range}_{rel}(x)$	Projection of binary relation x onto its second component

Figure 1: Operators on Composite Data Types

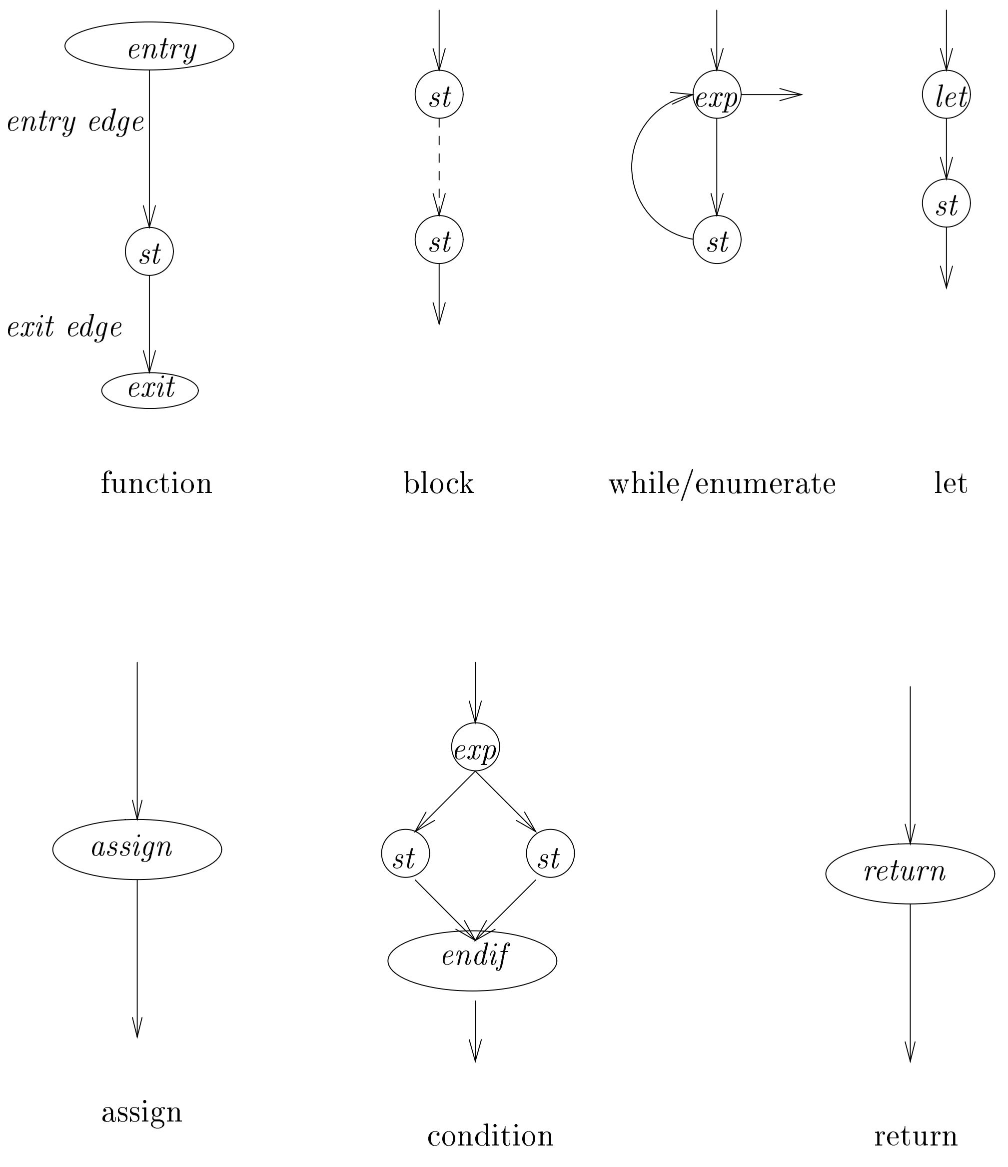

Figure 2: Program Graphs

```
    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
s2           (let (x = z.1, y = z.2)
s3             (Succ(x) ← Succ(x) with y;
s4              NumPred(y) ← NumPred(y) + 1);))
s5         let (MinEls = { })
s6           (enumerate v over S do
s7              if NumPred(v) = 0 then MinEls ← MinEls with v;
s8            while ¬(MinEls = { }) do
s9              let (a = arb(MinEls))
s10               (MinEls ← MinEls less a;
s11                result ← append(result, a);
s12                enumerate w over Succ(a) do
s13                  (NumPred(w) ← NumPred(w) − 1;
s14                   if NumPred(w) = 0 then MinEls ← MinEls with w))))
s15       return result
```

Figure 3: Topological Sorting Program

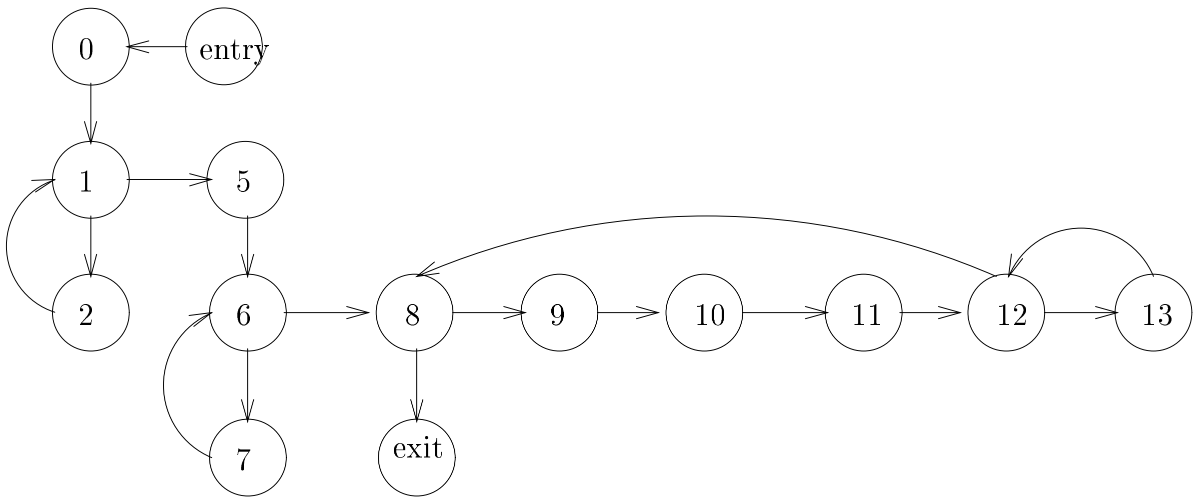

Figure 4: Graph Representation of Topological Sorting Program

3 Semantics

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

$$
\begin{array}{ll}
\textbf{Value} & \stackrel{\text{def}}{=} \textbf{Integer} + \textbf{Boolean} + \textbf{Symbol} + \cdots \\
\textbf{Environment} & \stackrel{\text{def}}{=} \textbf{Id} \to \textbf{Value} \\
\textbf{State} & \stackrel{\text{def}}{=} \textbf{Arc} \times \textbf{Environment}
\end{array}
$$

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 val: $\textbf{Exp} \to (\textbf{Environment} \to \mathcal{P}(\textbf{Value}))$. Since there is a non-deterministic operator arb, 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 $val[\![e]\!]r$. For most expressions, val is straightforward. For example

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

and

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

.

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

- If n is an assignment node representing the statement $v \leftarrow e$, then the next state is in $\{\langle\langle n, succ(n)\rangle, r \circ (v \to x)\rangle | x \in val[\![e]\!]r)\}$. If n is an assignment node representing the statement $v(d) \leftarrow e$, then the next state is in $\{\langle\langle n, succ(n)\rangle, r \circ (v \to val[\![v]\!]r \circ (x \to y))\rangle | x \in val[\![d]\!]r \wedge y \in val[\![e]\!]r\}$.

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

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

- If n is a let node representing the statement **let** $(v_1 = e_1, \cdots, v_l = e_l)$ s, then the next state is in $\{\langle\langle n, succ(n)\rangle, r \circ (v_i \to x_i)_{i=1}^{l}\rangle | \bigwedge_{i=1}^{l} x_i \in val[\![e_i]\!]r\}$.

- If n is an endlet node representing the statement **let** $(v_1 = e_1, \cdots, v_l = e_l)$ s, then the next state is $\langle\langle n, succ(n)\rangle, \{v \to r(v) | v \in \text{domain}(r) \wedge v \notin \{v_1, \ldots, v_l\}\}\rangle$.

- If n is a test node representing the statement **while** q **do** s, then the collection of possible next states contains $\langle\langle n, succ_t(n)\rangle, r\rangle$ if $true \in val[\![q]\!]r$ and $\langle\langle n, succ_f(n)\rangle, r\rangle$ if $false \in val[\![q]\!]r$.

- The semantics of the statement **enumerate** v **over** S **do** s is equivalent to that of the let statement

$$
\begin{array}{l}
\textbf{let } (S_{new} = S) \\
\quad \textbf{while } \neg(S_{new} = \{\,\}) \textbf{ do} \\
\quad\quad \textbf{let } (v = \text{arb}(S_{new})) \\
\quad\quad\quad (s; \\
\quad\quad\quad S_{new} \leftarrow S_{new} \text{ less } v).
\end{array}
$$

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

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

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

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

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:

$$x, x(y), x.1, x.2, \text{range}_{seq}(x), \text{domain}(x), \text{range}(x), \text{domain}_{rel}(x), \text{range}_{rel}(x), \text{power}(x)$$

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 α and γ such that (α, γ) forms an adjoined pair of functions: $\alpha \circ \gamma = \mathbf{identity}$ and $\gamma \circ \alpha \sqsupseteq \mathbf{identity}$. The *abstraction function* α 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* γ 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 $r \in \gamma(\alpha(c))$. $\square$

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{array}{lll}
\mathrm{range}_{seq}([\,]) & \Longrightarrow \{\,\} & \\
\mathrm{domain}(\{\!|\ |\!\}) & \Longrightarrow \{\,\} & \\
\mathrm{range}(\{\!|\ |\!\}) & \Longrightarrow \{\,\} & \\
\mathrm{notempty}(\{\,\}) & \Longrightarrow \mathit{false} & \\
\mathrm{notempty}(t_1 \text{ with } t_2) & \Longrightarrow \mathit{true} & \\
\mathrm{notempty}(t_1 \cup t_2) & \Longrightarrow \mathrm{notempty}(t_1) \vee \mathrm{notempty}(t_2) & \\
\neg(t = \{\,\}) & \Longrightarrow \mathrm{notempty}(t) & \\
t_1 \in t_2 & \Longrightarrow t_1 \in \mathrm{range}_{seq}(t_2) & t_2 \text{ has type } seq(\delta) \\
\mathrm{arb}(t_1) \in t_2 & \Longrightarrow \mathrm{notempty}(t_1) \wedge t_1 \subseteq t_2 & \\
p[y(x)] & \Longrightarrow x \in \mathrm{domain}(y) \wedge p(y(x)) & y \text{ has type } map(\delta, \sigma) \\
t_1 \in \mathrm{power}(t_2) & \Longrightarrow t_1 \subseteq t_2 & \\
(t_1 \cup t_2) \subseteq t_3 & \Longrightarrow (t_1 \subseteq t_3) \wedge (t_2 \subseteq t_3) & \\
t_1 \subseteq (t_2 \cap t_3) & \Longrightarrow (t_1 \subseteq t_2) \wedge (t_1 \subseteq t_3) & \\
t_1 \in (t_2 \text{ with } t_3) & \Longrightarrow (t_1 \in t_2) \vee (t_1 = t_3) & \\
t_1 \in (t_2 \text{ less } t_3) & \Longrightarrow (t_1 \in t_2) \wedge \neg(t_1 = t_3) & \\
(t_1 \text{ with } t_2) \subseteq t_3 & \Longrightarrow (t_1 \subseteq t_3) \wedge (t_2 \in t_3) & \\
t_1 \subseteq (t_2 \text{ less } t_3) & \Longrightarrow (t_1 \subseteq t_2) \wedge \neg(t_3 \in t_1) & \\
\mathrm{range}_{seq}(\mathrm{append}(t_1, t_2)) & \Longrightarrow \mathrm{range}_{seq}(t_1) \text{ with } t_2 & \\
\{v\colon v \in t\} & \Longrightarrow t & t \text{ has type } set(\delta) \\
\{v\colon v \in t\} & \Longrightarrow \mathrm{range}_{seq}(t) & t \text{ has type } seq(\delta) \\
[v\colon v \in t] & \Longrightarrow t & \\
\mathrm{domain}(m \circ (v \to e)) & \Longrightarrow \mathrm{domain}(m) \text{ with } v & \\
\mathrm{range}(m \circ (v \to e)) & \Longrightarrow (\mathrm{range}(m) \text{ less } m(v)) \text{ with } e & \\
\mathrm{domain}(\{\!|d \to r\colon d \in t|\!\}) & \Longrightarrow t & t \text{ has type } set(\delta) \\
\mathrm{domain}(\{\!|d \to r\colon d \in t|\!\}) & \Longrightarrow \mathrm{range}_{seq}(t) & t \text{ has type } seq(\delta) \\
t_1 = t_2 & \Longrightarrow t_1 \subseteq t_2 \wedge t_2 \subseteq t_1 & t_1, t_2 \text{ are sets}
\end{array}
$$

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:

Tautolgy

$$\frac{}{\{\}\subseteq x} \qquad \frac{}{x\subseteq x}$$

Empty Set

$$\frac{x\in y}{\text{notempty}(y)} \qquad \frac{\text{notempty}(x), x\subseteq y}{\text{notempty}(y)}$$

Subset

$$\frac{x\subseteq y, y\subseteq x}{x=y} \qquad \frac{x=y}{x\subseteq y, y\subseteq x} \qquad \frac{x\subseteq y, y\subseteq z}{x\subseteq z}$$

Power Set

$$\frac{x\subseteq y}{\text{power}(x)\subseteq\text{power}(y)} \qquad \frac{\text{power}(x)\subseteq\text{power}(y)}{x\subseteq y}$$

$$\frac{x\subseteq y}{x\in\text{power}(y)} \qquad \frac{x\in\text{power}(y)}{x\subseteq y}$$

Binary Relation

$$\frac{x\in y}{x.1\in\text{domain}_{rel}(y)} \qquad \frac{x\in y}{x.2\in\text{range}_{rel}(y)}$$

Map Range

$$\frac{x\in\text{domain}(y)}{y(x)\in\text{range}(y)} \qquad \frac{y(x)\in\text{range}(y)}{x\in\text{domain}(y)}$$

Equality

$$\frac{x=y, y=z}{x=z} \qquad \frac{x=y}{f(x)=f(y)}\textit{f any function in the language}$$

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 $\frac{x\subseteq y}{\text{power}(x)\subseteq\text{power}(y)}$ 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 $S_V^+ = S_V^*$.*

Proof (proof outline.) Suppose that there is a bound assertion $p\in S_V^*$ but $p\notin S_V^+$. We construct a model of S_V^+ in which p is false. This is a contradiction because $(S_V^+)^* = S_V^*$. 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. $\mathcal{A}_b(t)$ maps a set-valued expression t to a collection of bound terms that are subsets of t, while $\mathcal{A}_p(t)$ maps a set-valued expression t to a collection of bound terms that are supersets of t. In other words, for every $t_1 \in \mathcal{A}_b(t)$ and $t_2 \in \mathcal{A}_p(t)$, $t_1 \subseteq t$ and $t \subseteq t_2$ are valid. If t is a bound term, then $\mathcal{A}_b(t) = \mathcal{A}_p(t) = \{t\}$. 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 $\mathcal{A}_w$ and $\mathcal{A}_s$ are the bound-assertion abstraction functions to be defined in the next section:

$$
\begin{array}{ll}
\mathcal{A}_b(t_1 \text{ with } t_2) & \stackrel{\text{def}}{=} \mathcal{A}_b(t_1) \\
\mathcal{A}_b(t_1 \cup t_2) & \stackrel{\text{def}}{=} \mathcal{A}_b(t_1) \cup \mathcal{A}_b(t_2) \\
\mathcal{A}_b(t_1 \cap t_2) & \stackrel{\text{def}}{=} \mathcal{A}_b(t_1) \cap \mathcal{A}_b(t_2) \\
\mathcal{A}_b(\{v\colon p\}) & \stackrel{\text{def}}{=} \{x | \{v \in x\} \in \mathcal{A}_s(p)\} \\
\mathcal{A}_b(\text{range}_{seq}([v\colon p])) & \stackrel{\text{def}}{=} \{x | \{v \in x\} \in \mathcal{A}_s(p)\} \\
\mathcal{A}_b(\text{domain}(\{\!|d \to r\colon p|\!\})) & \stackrel{\text{def}}{=} \{x | \{d \in x\} \in \mathcal{A}_s(p)\} \\
\mathcal{A}_b(\text{range}(\{\!|d \to r\colon p|\!\})) & \stackrel{\text{def}}{=} \{x | \{r \in x\} \in \mathcal{A}_s(p)\}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{A}_p(t_1 \text{ less } t_2) & \stackrel{\text{def}}{=} \mathcal{A}_p(t_1) \\
\mathcal{A}_p(t_1 - t_2) & \stackrel{\text{def}}{=} \mathcal{A}_p(t_1) \\
\mathcal{A}_p(t_1 \cup t_2) & \stackrel{\text{def}}{=} \mathcal{A}_p(t_1) \cap \mathcal{A}_p(t_2) \\
\mathcal{A}_p(t_1 \cap t_2) & \stackrel{\text{def}}{=} \mathcal{A}_p(t_1) \cup \mathcal{A}_p(t_2) \\
\mathcal{A}_p(\{v\colon p\}) & \stackrel{\text{def}}{=} \{x | (v \in x) \in \mathcal{A}_w^+(p)\} \\
\mathcal{A}_p(\text{range}_{seq}([v\colon p])) & \stackrel{\text{def}}{=} \{x | (v \in x) \in \mathcal{A}_w^+(p)\} \\
\mathcal{A}_p(\text{domain}(\{\!|d \to r\colon p|\!\})) & \stackrel{\text{def}}{=} \{x | (d \in x) \in \mathcal{A}_w^+(p)\} \\
\mathcal{A}_p(\text{range}(\{\!|d \to r\colon p|\!\})) & \stackrel{\text{def}}{=} \{x | (r \in x) \in \mathcal{A}_w^+(p)\}
\end{array}
$$

5.4 Bound Assertion Abstraction

We define two abstraction functions that map boolean expressions to logically related bound assertions. $\mathcal{A}_w(p)$ maps a boolean expression p to a collection of bound assertions that are logically weaker than p. In other words, for every $q \in \mathcal{A}_w(p)$, $p \to q$ is valid. Likewise, $\mathcal{A}_s(p)$ maps a boolean expression p to a collection of collections of bound assertions that are logically stronger than p. In other words, for every $S \in \mathcal{A}_s(p)$, $\bigwedge_{q \in S} q \to p$ is valid. If p is a bound assertion, then $\mathcal{A}_w(p) = \{p\}$ and $\mathcal{A}_s(p) = \{\{p\}\}$. 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:

$$
\begin{array}{lll}
\mathcal{A}_w(p \wedge q) & \stackrel{\text{def}}{=} \mathcal{A}_w(p) \cup \mathcal{A}_w(q) & \\
\mathcal{A}_w(p \vee q) & \stackrel{\text{def}}{=} \mathcal{A}_w(p) \cap \mathcal{A}_w(q) & \\
\mathcal{A}_w(\text{notempty}(t)) & \stackrel{\text{def}}{=} \{\text{notempty}(x) | x \in \mathcal{A}_p(t)\} & \\
\mathcal{A}_w(x_1 \in t_2) & \stackrel{\text{def}}{=} \{x_1 \in x_2 | x_2 \in \mathcal{A}_p(t_2)\} & x_1 \text{ is a bound term} \\
\mathcal{A}_w(t_1 \subseteq t_2) & \stackrel{\text{def}}{=} \{x_1 \subseteq x_2 | x_1 \in \mathcal{A}_b(t_1) \wedge x_2 \in \mathcal{A}_p(t_2)\} & \\
\mathcal{A}_w(t_1 = \text{arb}(t_2)) & \stackrel{\text{def}}{=} \mathcal{A}_w(\text{notempty}(t_2) \wedge t_1 \in t_2) & \\
\mathcal{A}_w(t_1 = \text{first}(t_2)) & \stackrel{\text{def}}{=} \mathcal{A}_w(\text{notempty}(\text{range}_{seq}(t_2)) \wedge t_1 \in \text{range}_{seq}(t_2)) & \\
\mathcal{A}_w(t_1 = t_2(t_3)) & \stackrel{\text{def}}{=} \mathcal{A}_w(t_3 \in \text{domain}(t_2) \wedge t_1 \in \text{range}(t_2)) & t_2 \text{ has type } map(\delta, \sigma) \\
\mathcal{A}_w(t_1 = t_2) & \stackrel{\text{def}}{=} \mathcal{A}_w(\text{range}_{seq}(t_1) \subseteq \text{range}_{seq}(t_2)) & \\
 & \quad \cup \mathcal{A}_w(\text{range}_{seq}(t_2) \subseteq \text{range}_{seq}(t_1)) & \text{sequence equality} \\
\mathcal{A}_w(t_1 = t_2) & \stackrel{\text{def}}{=} \mathcal{A}_w(\text{domain}(t_1) \subseteq \text{domain}(t_2)) & \\
 & \quad \cup \mathcal{A}_w(\text{domain}(t_2) \subseteq \text{domain}(t_1)) & \\
 & \quad \cup \mathcal{A}_w(\text{range}(t_1) \subseteq \text{range}(t_2)) & \\
 & \quad \cup \mathcal{A}_w(\text{range}(t_2) \subseteq \text{range}(t_1)) & \text{map equality}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{A}_s(p \wedge q) & \stackrel{\text{def}}{=} \{x_1 \cup x_2 | x_1 \in \mathcal{A}_s(p) \wedge x_2 \in \mathcal{A}_s(q)\} & \\
\mathcal{A}_s(p \vee q) & \stackrel{\text{def}}{=} \mathcal{A}_s(p) \cup \mathcal{A}_s(q) & \\
\mathcal{A}_s(\text{notempty}(t)) & \stackrel{\text{def}}{=} \{\{\text{notempty}(x)\} | x \in \mathcal{A}_b(t)\} & \\
\mathcal{A}_s(x_1 \in t_2) & \stackrel{\text{def}}{=} \{\{x_1 \in x_2\} | x_2 \in \mathcal{A}_b(t_2)\} & x_1 \text{ is a bound term} \\
\mathcal{A}_s(t_1 \subseteq t_2) & \stackrel{\text{def}}{=} \{\{x_1 \subseteq x_2\} | x_1 \in \mathcal{A}_p(t_1) \wedge x_2 \in \mathcal{A}_b(t_2)\} &
\end{array}
$$

5.5 Example

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

$$
\begin{array}{l}
\text{domain}(Succ) \subseteq S \\
S \subseteq \text{domain}(Succ) \\
\text{range}(Succ) \subseteq \text{range}(\{\!|v \to \{\,\}\colon v \in S|\!\}) \\
\text{range}(\{\!|v \to \{\,\}\colon v \in S|\!\}) \subseteq \text{range}(Succ)
\end{array}
$$

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 $\text{range}(Succ) \subseteq \text{power}(S)$, because bound assertion

$\{\} \in \mathrm{power}(S)$ is in $\mathcal{A}_w^+(v \in S)$. Finally, no bound assertion is abstracted by $\mathcal{A}_w$ from the last expression.

5.6 Correctness

Theorem 3 *For every bound term $t_2 \in \mathcal{A}_b(t_1)$, $t_2 \subseteq t_1$ 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 $q \in \mathcal{A}_w(p)$, $p \to q$ is valid. For every collection of bound assertions $S \in \mathcal{A}_s(p)$, $\bigwedge_{q\in S} q \to p$ 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 $v \in t \to p$ 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 $\mathcal{A}_p$. It follows that $t_1 \subseteq t_2 \to q$ is valid. For $q \in \mathcal{A}_w(t_1 = \mathrm{arb}(t_2))$, $q \in \mathcal{A}_w(\mathrm{notempty}(t_2))$ or $q \in \mathcal{A}_w(t_1 \in t_2)$ by definition. By induction hypothesis on $\mathcal{A}_w$, $\mathrm{notempty}(t_2) \to q$ or $t_1 \in t_2 \to q$ is valid. In other words, $\mathrm{notempty}(t_2) \wedge t_1 \in t_2 \to q$ is valid. Hence $t_1 = \mathrm{arb}(t_2) \to q$ is valid since $t_1 = \mathrm{arb}(t_2) \to \mathrm{notempty}(t_2) \wedge t_1 \in t_2$ is valid. Other cases for $\mathcal{A}_w$ are similar.

For collection of bound assertions $S \in \mathcal{A}_s(p \wedge q)$, 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 $\mathcal{A}_s$, $\bigwedge_{r\in S_1} r \to p$ and $\bigwedge_{r\in S_2} r \to q$ are valid. Hence $\bigwedge_{r\in S} r \to (p \wedge q)$ is valid. Other cases for $\mathcal{A}_s$ are similar. □

6 Interpretation Function

The abstract interpretation $\Im$ 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, $\Im$ computes post-conditions of bound assertions over syntactic constructs. If $c' = \emptyset$, then the post-condition is *true*. If node n has only one outgoing arc, then $\Im(c, n)$ stands for $\Im(c, \langle n, \mathit{succ}(n)\rangle)$. If node n represents statement s, then $\Im(c, s)$ stands for $\Im(c, n)$. For notational convenience, $\Im(S, n)$ denotes $\Im(S^+, n)$ for set of bound assertions S, and $\Im(p, n)$ denotes $\Im(\{p\}, n)$ 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.

$$
\begin{array}{ll}
\Im(c, v \leftarrow e) & \stackrel{\text{def}}{=} (S \cup \{p | p \in c \wedge (\exists c')(c' \subseteq c \wedge c' \in \mathcal{A}_s(p[e/v]))\})^+ \\
& \quad\quad S = \{\,\} \text{ if } v \text{ occurs in } e \\
& \quad\quad S = \mathcal{A}_w(v = e) \text{ otherwise} \\
\Im(c, v(d) \leftarrow e) & \stackrel{\text{def}}{=} (S \cup \{p | p \in c \wedge (\exists c')(c' \subseteq c \wedge c' \in \mathcal{A}_s(p[v \circ (d \rightarrow e)/v]))\})^+ \\
& \quad\quad S = \{\,\} \text{ if } v \text{ occurs in } d \text{ or } e \\
& \quad\quad S = \mathcal{A}_w(v(d) = e) \text{ otherwise} \\
\Im(c, \langle n, succ_t(n)\rangle) & \stackrel{\text{def}}{=} (c \cup \mathcal{A}_w(q))^+ \\
\Im(c, \langle n, succ_f(n)\rangle) & \stackrel{\text{def}}{=} (c \cup \mathcal{A}_w(\neg q))^+ \\
\Im(c, \textbf{let } v_1 = e_1, \cdots, v_l = e_l) & \stackrel{\text{def}}{=} (c \cup \mathcal{A}_w(v_1 = e_1 \wedge \cdots \wedge v_l = e_l))^+ \\
\Im(c, \textbf{if } q \textbf{ then } s_1 \textbf{ else } s_2) & \stackrel{\text{def}}{=} \Im((c \cup \mathcal{A}_w(q))^+, s_1) \cap \Im((c \cup \mathcal{A}_w(\neg q))^+, s_2) \\
\Im(c, (s_1; \ldots; s_l)) & \stackrel{\text{def}}{=} \Im(\Im(c, s_1), (s_2; \ldots; s_l)) \\
\Im(c, \textbf{let } (v_1 = e_1, \cdots, v_l = e_l)\ s) & \stackrel{\text{def}}{=} \{p | p \in \Im((c \cup \mathcal{A}_w(v_1 = e_1 \wedge \cdots \wedge v_l = e_l))^+, s) \\
& \quad\quad \wedge p \text{ is free of } v_1, \ldots, v_l\}
\end{array}
$$

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 $\Im(c, s)$ logically implies c.

An *invariant* of while statement **while** q **do** s is an abstract context Inv such that $Inv \subseteq \Im((Inv \cup \mathcal{A}_w(q))^+, s)$. 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 **while** q **do** s. For abstract context c, let Inv be the largest invariant of the statement such that $Inv \subseteq c$:

$$
\begin{array}{ll}
\Im(c, \langle n, succ_t(n)\rangle) & \stackrel{\text{def}}{=} (Inv \cup \mathcal{A}_w(q))^+ \\
\Im(c, \textbf{while } q \textbf{ do } s) & \stackrel{\text{def}}{=} (Inv \cup \mathcal{A}_w(\neg q))^+
\end{array}
$$

The first rule for enumerate statements of the form **enumerate** v **over** S **do** s is the same if we replace q by $v \in S$. The second rule is specified as follows, since v should not be visible outside the statement:

$\Im(c, \textbf{enumerate } v \textbf{ over } S \textbf{ do } s) \stackrel{\text{def}}{=} \mathit{Inv}$

6.3 Example

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

p_1: $\text{domain}(\mathit{Succ}) = S$
p_2: $\text{range}(\mathit{Succ}) \subseteq \text{power}(S)$
p_3: $\text{domain}_{rel}(R) \subseteq S$
p_4: $\text{range}_{rel}(R) \subseteq S$

through the first enumerate statement s_1:

s_1: **enumerate** z **over** R **do**
s_2: **let** $(x = z.1, y = z.2)$
s_3: $(\mathit{Succ}(x) \leftarrow \mathit{Succ}(x)$ with y;
s_4: $\mathit{NumPred}(y) \leftarrow \mathit{NumPred}(y) + 1)$;

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

$$\begin{aligned}
&\mathcal{A}_s(p_1[\mathit{Succ} \circ (x \to \mathit{Succ}(x) \text{ with } y)/\mathit{Succ}]) \\
&\quad = \mathcal{A}_s(\text{domain}(\mathit{Succ}) \text{ with } x \subseteq S \wedge S \subseteq \text{domain}(\mathit{Succ}) \text{ with } x) \\
&\quad = \{\{\text{domain}(\mathit{Succ}) \subseteq S, x \in S\} \cup \{S \subseteq \text{domain}(\mathit{Succ})\}\} \\
&\quad = \{\{p_1, x \in S\}\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{A}_s(p_2[\mathit{Succ} \circ (x \to \mathit{Succ}(x) \text{ with } y)/\mathit{Succ}]) \\
&\quad = \mathcal{A}_s(\text{range}(\mathit{Succ}) \text{ less } \mathit{Succ}(x) \subseteq \text{power}(S) \wedge \mathit{Succ}(x) \subseteq S \wedge y \in S) \\
&\quad = \{\{\text{range}(\mathit{Succ}) \subseteq \text{power}(S)\} \cup \{\mathit{Succ}(x) \subseteq S, y \in S\}\} \\
&\quad = \{\{p_2, \mathit{Succ}(x) \subseteq S, y \in S\}\}
\end{aligned}$$

both of which contain subcollections of $\Im(P', n)$. 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, $P \subseteq \Im(P', s_2)$. In conclusion, all bound assertions in P are loop invariants of the enumerate statement s_1 and can be propagated through it: $\Im(P, s_1) = P$.

6.4 Correctness

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

Proof (proof outline.) $\Im$ 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 $\Im(c, a)$ 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 $\Im(c, v \leftarrow e)$, if $p \in \mathcal{A}_w(v = e)$ then p is true after the assignment because $v = e$ is. Otherwise there exists $c' \subseteq c$ true before the assignment such that $c' \in \mathcal{A}_s(p[e/v])$. 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 $\mathit{Inv} \subseteq c$. If *Inv* and q are true before executing s', then $\Im((\mathit{Inv} \cup \mathcal{A}_w(q))^+, s')$ is true after executing s' by induction hypothesis, and hence *Inv* is true after executing s' since $\mathit{Inv} \subseteq \Im((\mathit{Inv} \cup \mathcal{A}_w(q))^+, s')$. Therefore $\Im(c, s) = \mathit{Inv}$ is true after executing s since c is true before executing s and $\mathit{Inv} \subseteq c$. □

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:

$\mathrm{domain}_{rel}(R) \subseteq S$
$\mathrm{range}_{rel}(R) \subseteq S$

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

$\mathrm{range}_{seq}(\mathit{result}) \subseteq S$
$\mathrm{domain}(\mathit{Succ}) = S$
$\mathrm{range}(\mathit{Succ}) \subseteq \mathrm{power}(S)$
$\mathrm{domain}(\mathit{NumPred}) = S$

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 $\mathit{MinEls} \subseteq S$ is obtained. Lets denote these bound assertions by P. Similar to Section 6.3 we can push P through the first statement of s_6. Notice that $\mathcal{A}_w(\neg(\mathit{MinEls} = \{\,\})) = \{\text{notempty}(\mathit{MinEls})\}$. In order to push P through s_8, we have to compute $\Im((P \cup \{\text{notempty}(\mathit{MinEls})\})^+, s_9)$. After pushing $P \cup \{\text{notempty}(\mathit{MinEls})\}$ through the let node of s_9, we get new bound assertions $a \in \mathit{MinEls}, a \in S$, 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 $\mathcal{A}_w(w \in \mathit{Succ}(a)) = \{a \in \text{domain}(\mathit{Succ}), w \in \mathit{Succ}(a)\}$, and $w \in S$ is in the closure of $P' \cup \{a \in \text{domain}(\mathit{Succ}), w \in \mathit{Succ}(a)\}$. Similar to Section 6.4, we can push P' through s_{13}, hence $\Im(P', s_{12}) = P'$. Finally, P can be pushed through s_9, therefore $\Im(P, s_8) = P$. All bound assertions in P not mentioning *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 *MinEls*, a domain bound and an access bound for *Succ* and *NumPred*, and a range bound for *result*. In addition, power(S) is a range bound for *Succ*.

An efficient implementation of topsort requires that access to the maps *NumPred* and *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 *NumPred*, and *Succ*. Elements of S appearing in *MinEls*, range(*Succ*), R, and *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

[1] Ammarguellat, Z., Harrison III, W., "Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation"; *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, 283-295.

[2] Cousot, P., "Semantic Foundations of Program Analysis"; *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones (editors), Prentice-Hall, 1981, 303-342.

[3] Cousot, P., Cousot, R., "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximations of Fixpoints"; *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, January 1977, 238-252.

[4] Goldberg, A., Kotik, G., "Knowledge-Based Programming: An Overview of Data Structure Selection and Control Structure Refinement"; *Technical Report* KES.U.83.7, Kestrel Institute, November 1983.

[5] Goldberg, A., Smith, D., "Towards a Performance Estimation Assistant"; *Technical Report* KES.U.86.10, Kestrel Institute, November 1986.

[6] Hoare, C., "An Axiomatic Basis for Computer Programming"; *Communications of the ACM* **12**:10, 1969, 576-583.

[7] Krivine, J. *Introduction to Axiomatic Set Theory*, Reidel Publishing Co., Holland, 1971.

[8] Manna, Z., Waldinger, R., *The Logical Basis for Computer Programming, Vol.2: Deductive Systems*, Addison-Wesley, 1990.

[9] Abramsky, S., Hankin, C. (editors), *Abstract Interpretation of Declarative Languages*, Ellis Horwood Limited, 1987.

[10] Qian, X., Goldberg, A., "Bounds Analysis by Abstract Interpretation," Kestrel Institute Report, 1994.

[11] Schwartz, J., "Optimization of Very High Level Languages—I: Value Transmission and Its Corollaries"; *Computer Languages* **1**, 1975, 161-194.

[12] Schwartz, J., "Optimization of Very High Level Languages—II: Deducing Relationships of Inclusion and Membership"; *Computer Languages* **1**, 1975, 197-218.