Automatic Generation and Management of Interprocedural Program Analyses*

Kwangkeun Yi and Williams Ludwell Harrison III
(kwang|harrison)@csrd.uiuc.edu
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
465 CSRL, 1308 West Main St., Urbana, IL 61801-2307

Abstract

We have designed and implemented an interprocedural program analyzer generator, called *system Z*. Our goal is to automate the generation and management of semantics-based interprocedural program analysis for a wide range of target languages.

System Z is based on the abstract interpretation framework. The input to system Z is a high-level specification of an abstract interpreter. The output is a C code for the specified interprocedural program analyzer. The system provides a high-level command set (called *projection expressions*) in which the user can tune the analysis in accuracy and cost. The user writes projection expressions for selected domains; system Z takes care of the remaining things so that the generated analyzer conducts an analysis over the projected domains, which will vary in cost and accuracy according to the projections.

We demonstrate the system's capabilities by experiments with a set of generated analyzers which can analyze C, FORTRAN, and SCHEME programs.

1 Introduction

Semantic analysis, and especially interprocedural analysis, is an important component of compilers for modern computer systems. Our goal is to automate the generation of such global program analyses and shorten the cycle time of designing, experimenting, and redesigning. We have designed and implemented the *system Z* which automates the generation and management of interprocedural program analyses. The system is based on the abstract interpretation framework [6, 7]. The user specifies an abstract interpreter in a specification language. The system compiles the specification into a C code, which becomes an executable interprocedural program analyzer when linked with a target language parser. The specification language has a high-level command set (called *projection expressions*) by which the user can tune the analysis in accuracy and cost. By this facility, he can quickly try several design choices and experiment with the resulting analyses. The specification language is general enough to express a large class of global analyses for a wide range of target languages, and abstract enough to specify an analysis succinctly.

Similar tools [18, 14, 8, 24] have been reported in the literature, but they have several shortcomings relative to system Z. They do not support interprocedural analysis specification, are limited to specific target languages and a class of simple analyses (e.g. bit-vector representation of the data flow values), require substantial amount of code from the user, and have no high-level facility to tune the analysis in cost and accuracy. Data flow analysis specifications in these systems are similar to production rules in YACC. They add analysis actions to an attribute grammar and use a parsing procedure or a fixed number of traversals of an attribute syntax tree to collect data flow information. All the value structures and semantic actions at each attribute grammar node must be provided in the implementation language by the user. Recently, Tjiang and Hennessy [21] reported a similar tool to generate an analyzer for a low-level intermediate language, with C++ as its specification language. An interesting feature of this tool is that its fixpoint computation procedure can be simplified by means of flow graph reduction rules specified by the user. It is unclear, however, how this feature can be used for the case where the flow graph of a program cannot be determined before the analysis, which is the case for languages in which a function call expression need not specify (syntactically) which function it calls. Young [26] reported a library that he used to implement several semantic analyses. Steffen [19] reported a specification framework that uses modal logic formulae to specify data flow analysis algorithms. The work most similar to ours is by Venkatesh [22]. His specification language allows denotational semantics to be augmented with a collecting mechanism for program analysis. None of these tools provides a high-level facility to tune the analysis in accuracy and cost.

1.1 System Overview

The overall configuration of system Z is shown in Figure 1. The input to system Z is an *abstract interpreter* specification for a target language. The user normally takes the following steps in arriving to the specification of the abstract interpreter. He starts from a *standard interpreter* of a target language. A standard interpreter has nothing beyond what is necessary to interpret a program. To the standard interpreter, he adds mechanisms to record data flow information of interest, resulting in an *instrumented interpreter*. As the last step, he approximates the instrumented interpreter into an *abstract interpreter*. The abstract interpreter is suitable for use as a compile-time analysis because it allows a simplified, approximate interpretation, and because the induced program analysis is guaran-

*This work was supported in part by the **U.S. Department of Energy** under Grant No. DE-FG02-85ER25001 with additional support from **NSF** under Grant No. NSF CCR 90-24554.

To appear in **The Twentieth Annual ACM Symposium on Principles of Programming Languages**, January, 1993.

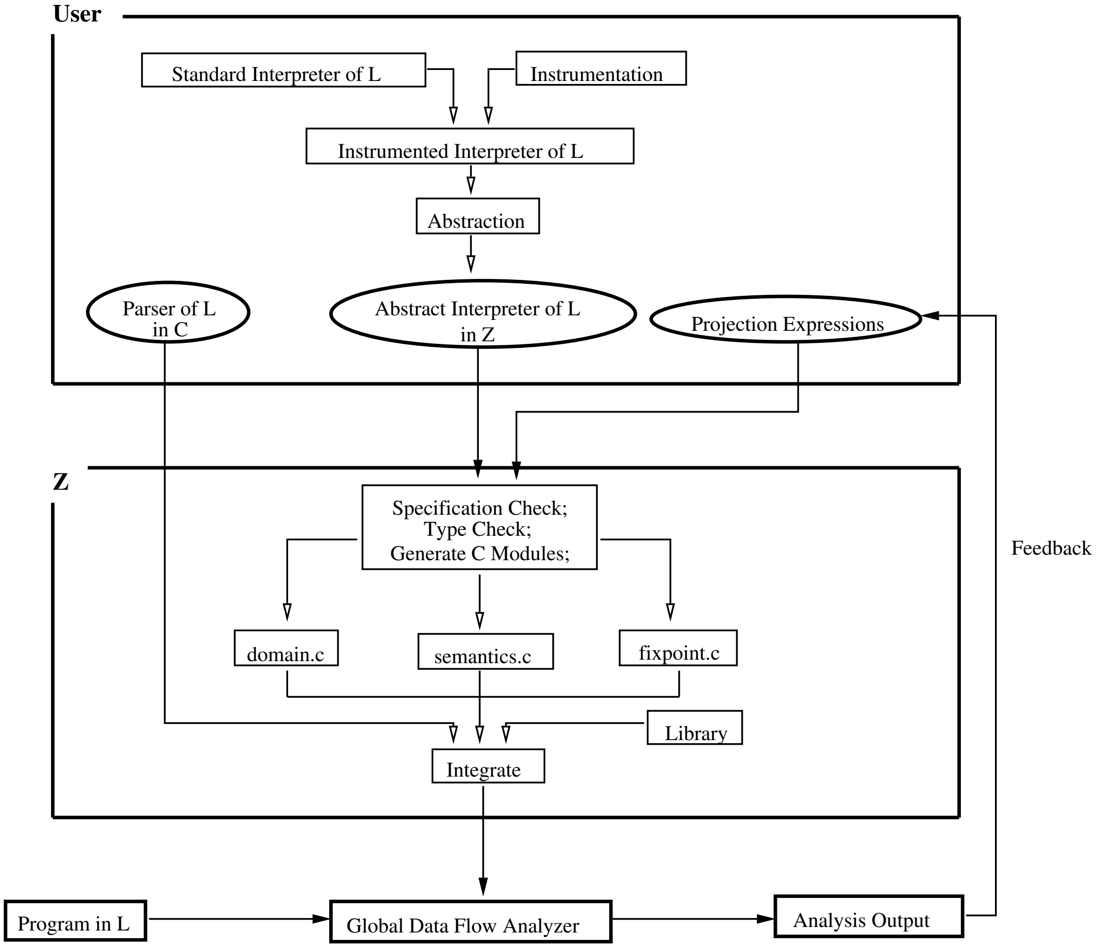

Figure 1: The System Z

teed always to terminate. This abstract interpreter specification is the starting point for (input to) system Z.

From an abstract interpreter specification, system Z generates three C modules: one for domain element management, one for semantics operations, and one for fixpoint computation. These three modules are linked with libraries and the target language parser into an executable program analyzer. The input to the generated analyzer is a program in the target language. The output of the generated analyzer is the result of the specified analysis of the program. The analysis result is a table that maps each program point to information which describes the program states that can occur at that point during execution. Depending on a command option, the analysis result (in binary form) is dumped into a file, to be restored for subsequent examination. The system provides all necessary run-time support (e.g. garbage collector) and tools to trace the resulting analyses (e.g. data type pretty printer).

After experiments with a generated analyzer, its cost and accuracy behavior is observed. An obvious way to modify the analysis is to rewrite the entire specification again: to define new domains and to change the interpreter functions accordingly. System Z provides a simpler facility: *projection expressions* over the domains. The user writes projection expressions for selected domains. System Z then automatically generates an analysis which will process input programs over the simplified domain structure. This suggests the following approach to designing and experimenting with an analysis. The user defines one detailed abstract interpreter at the outset. Later, he write projections for selected domains. The system takes care of the remaining things so that the generated analyzer conducts an analysis over the projected domains, which will vary in cost and accuracy according to the projections.

1.2 Why Abstract Interpretation?

The abstract interpretation framework [6, 7, 1] proposes that designing a program analysis is equivalent to defining an interpreter for the target language. The virtues of system Z comes from the power of the abstract interpretation framework.

- *The abstract interpretation framework solves problems that compiler designers currently face.*

  Conventional data flow analysis methods (as summarized in [17, 12]) have several limitations. First, they are restricted to languages like FORTRAN where a program's flow graph can be (largely) determined from its text. Second, only with difficulty can they be adapted to constructs that require detailed semantic consideration. In languages like C or SCHEME a procedure is not always invoked by its defined name, and not all accessible memory locations are named explicitly when they are accessed. For example, in

```
f(g) = ···g(*x)···
```

`g` can be bound to several different functions. Unless we simulate (at compile-time) the parameter binding mechanism, `g`'s flow graph must be connected to every function, which will cause expensive and inaccurate analysis. The variable `x` does not (syntactically) specify to which memory location it points, because of unrestrained pointer manipulations, aliases, or dynamically generated memory. Unless we interpret (again at compile-time) those program constructs, we must assume that `x` points to every memory location.

The abstract interpretation framework addresses these limitations. It views an analysis as an abstract simulation of input programs. It is driven by a formal semantics of the target language. By forcing optimizing compiler designers to work rigorously with the semantics of input programs, the framework helps one to handle a detailed semantic analysis without much difficulty, and to keep complicated analyses and radical transformations from violating the original semantics of input programs.

- *The abstract interpretation framework does not differentiate interprocedural analysis from intra-procedural.*

  Under abstract interpretation, interprocedural analysis is no more difficult than intra-procedural analysis. An approximate, compile-time interpretation rule for procedure call expression is all that is needed for interprocedural analysis. As defined in this rule, the analysis will bind the parameters and interpret the body. This approach is different from conventional ones [5, 15, 23, 2] in two aspects. First, aliases (due to reference parameters) and recursive calls, which were considered two principal problems of interprocedural analysis, are treated by simulating directly the language construct that causes them (in this case, call expression). Second, the program's call graph is not assumed as given prior to the analysis. As we discussed above, this is important when the target language allows procedures as first-class objects. Accuracy improvement by clever representations of the call graph [5, 15] may still be applicable in abstract interpretation, if the improvement is applied as a post-analysis process, when the call graph has been constructed.

- *The abstract interpretation framework makes it feasible to generate compile-time analyses from high-level specifications automatically.*

  Designing a program analysis is equivalent to designing an abstract interpreter. Thus an appropriate formalism for interpreter specification provides a comfortable ground for automatic generation of a global program analysis. In denotational semantics [20], abstract interpreters are defined by domain equations together with semantics equations over the domains. For a useful class of domains these specifications can automatically be made executable. When linked with a fixpoint algorithm, this gives a program analyzer.

- *The abstract interpretation framework suggests simple ways to tune the analyses.*

  In the abstract interpretation framework, the efficiency and accuracy of an analysis is determined largely by the degree of *abstraction* of the analysis. This means that a command to control the degree of abstraction is a good specification method for tuning the analysis. The more abstract the interpreter, the less accurate the analysis will be, but the less time and space it will cost. The designer can measure the gain in information accuracy against the cost of the analysis.

Note that the abstract interpretation approach does not necessarily mean more accurate analysis than the conventional data flow approach. For languages and analysis problems to which the conventional approach is well suited, it can give an equally accurate and faster analysis than a corresponding abstract interpretation. What we claim is that abstract interpretation is needed to achieve the results of data flow analysis for more difficult and dynamic language constructs. By abstract interpretation, we can design comfortably a correct, detailed semantic analysis (whether it be interprocedural or intra-procedural) for a broad class of target languages.

1.3 Organization

In Section 2, we discuss how an abstract interpreter definition is understood by system Z as a specification for a program analysis. In Section 3, we present the abstract interpreter specification language Z. In this section we introduce projection expressions. In Section 4, we demonstrate the system's capabilities by experiments with generated analyzers for ANSI C, FORTRAN, and SCHEME programs. We show how projection expressions can change an analysis in accuracy and cost. We also present the performance of generated analyzers for a set of C and FORTRAN programs. In Section 5, we discuss future work and conclude. In Appendix A, we include parts of the abstract interpreter definitions used for the experiments in Section 4.

2 Program Analysis From an Abstract Interpreter Definition

System Z generates a global program analysis from a high level specification of an abstract interpreter. In this section, we show how the system compiles an abstract interpreter definition into a corresponding program analysis. The user must understand this process in order to specify an analysis that behaves as he intends.

2.1 Collecting Analysis Induced by an Abstract Interpreter

Z generates a program analyzer which computes, for each program point, information which describes the possible program states at that point during execution. We call this a *collecting analysis* [13].

How can an interpreter definition be a specification of a collecting analysis? An interpreter is a function that defines, for each language construct, its evaluation rule. Each evaluation rule is a state transformer: a function from a pre-state to a post-state. An abstract interpreter $\overline{I}$ of a target language is thus defined as a function of type

$$\overline{I}: \Sigma \rightarrow \overline{X} \rightarrow \overline{Y}$$

where Σ is the set of program points, $\overline{X}$ and $\overline{Y}$ are lattices of pre-states and post-states, respectively. An abstract interpreter definition looks like:

$$\begin{array}{l} \overline{I} = \lambda\sigma.\lambda\overline{x}.\ \text{case}(\sigma)\ \text{of} \\ \qquad \sigma\ \text{an assignment: } \overline{I}_1(\sigma, \overline{x}) \\ \qquad \sigma\ \text{a procedure call: } \overline{I}_2(\sigma, \overline{x}) \\ \qquad \ldots \\ \qquad \sigma\ \text{a constant: } \overline{I}_n(\sigma, \overline{x}) \end{array}$$

where $\overline{I}_i$'s usually involve recursive calls of $\overline{I}$.

The program points of a program P are the nodes of P's *abstract syntax tree*. The nodes of an abstract syntax tree are the language constructs as defined in abstract syntax of the language. Let Σ_P be the set of program points of a program P.

The collecting analysis of a program P from the interpreter definition $\overline{I}$ is the computation of

$$Tabulate(F_{\overline{I}}, \Sigma_P, \overline{x}_0) \quad \text{(see Figure 2)},$$

where $F_{\overline{I}}$ is the associated functional[1] of the recursive definition of $\overline{I}$ and $\overline{x}_0$ is the starting pre-state of the program. The analysis result is two tables T_X and T_Y. The tables have, for each program point σ, a pair of a pre-state $T_X(\sigma) \in \overline{X}$ and a post-state $T_Y(\sigma) \in \overline{Y}$ that describe states that occur before and after that point during execution. Figure 2 shows a basic algorithm to compute the collecting analysis. In reality, we use an optimized version which iterates only for a subset of program points whose T_X and T_Y entries were changed by the previous iteration.

Note that by means of a separate fixpoint computation driver ($Tabulate$), an abstract interpreter definition, which include neither a fixpoint operator nor a property-collecting mechanism, becomes a collecting analysis specification.

2.2 Designing an Abstract Interpreter

It is the user's job to define an abstract interpreter for input to system Z. To define an abstract interpreter is to define abstract value spaces and to write, for each language construct, an abstract evaluation rule that operates over the abstract values. One question is: how do we write a correct abstract interpreter? This question reduces to: what does each abstract value mean? The denotational semantics formalism provides us with an answer to this question [4, 3].

In denotational semantics [20], *domains* are used to give mathematical meaning to types. In the abstract interpreters given to Z, a data value is understood as denoting an *ideal* of the corresponding domain in the concrete, instrumented interpreter. (The concrete, instrumented interpreter is defined in the denotational semantics formalism.) An ideal J of a domain is a subset which is downward closed ($y \sqsubseteq x \in J$ implies $y \in J$) and upward complete (every chain of J has its least upper bound in J). The relation between an abstract value $\overline{x}$ and its meaning $\gamma(\overline{x})$ is described by the adjoined pair:

$$\begin{array}{rcl} \gamma(\overline{x}) & = & \{\, x \mid \alpha(x) \sqsubseteq \overline{x} \,\} \\ \alpha(x) & = & \sqcap\{\, \overline{x} \mid x \in \gamma(\overline{x}) \,\}. \end{array}$$

Each space of abstract data values is structured into a finite-height lattice ordered by set inclusion of ideals. The finite-height requirement guarantees the termination of the induced analysis.

Each evaluation rule of an abstract interpreter is required to be monotonic: it returns no lower a value when given a higher input. This requirement, with the finite-height of lattices, guarantees the termination of the induced program analysis. In addition, every abstract operation $\overline{f}$ must be a upper approximation of the corresponding f in the instrumented interpreter:

$$f(x) \in \gamma(\overline{f}(\overline{x})) \quad \forall x \in \gamma(\overline{x}).$$

This requirement guarantees the correctness of an abstract interpreter with respect to its concrete correspondent.

System Z provides a set of constructs to declare abstract domains and operations to manipulate abstract domain elements. *The user is responsible for defining a correct abstract interpreter in the sense that we have discussed above.* The concrete, instrumented interpreter is neither seen nor processed by system Z.

3 Specification Language of the Abstract Interpreter

In this section, we present the specification language Z in which we specify an abstract interpreter. We give the language the same name as the system. An abstract interpreter specification in Z is compiled into a C program that computes the specified collecting analysis as outlined in Section 2.

Z is a strongly-typed applicative language with user-defined types. Expressions are evaluated eagerly (that is, Z is not a lazy language). A user-defined type is either a set or a domain. A domain is a finite-height lattice. The user can define an enumerated set, an integer range set, an index set, or a product set. Similarly, the user can define a lifted domain, a powerset domain, a product domain, and a function domain. Z requires that the domains have finite height. Together with monotonic operations over the domains, this guarantees the termination of the specified program analysis. This requirement implies that Z cannot accept reflexive type definitions. A reflexive type is one that is defined recursively. A reflexive type results in an infinite-height lattice, which is unacceptable for compile-time analysis.

Interestingly, the language Z does not provide a user-visible fixpoint operator. As we will see, the absence of an embedded fixpoint operator, the absence of reflexive domains, and its eager evaluation semantics do not restrict significantly the language constructs that Z is able to analyze.

One interesting capability of Z is a high-level facility to tune the analysis in cost and accuracy. This is done by *projection expressions* applied to abstract domains. The user writes projection expressions for selected domains; system Z then generates an analysis which will analyze over the simplified domain structures.

To present the language Z, we will show side by side two forms of the language. One is its real look and the other is its definition in conventional mathematical notation. We use `typewriter` style for reserved words of Z, and *italic* style for nonterminal symbols. Among nonterminal symbols, "x" is used for bound names, and "τ" for types (domains or sets), specifically, "D" for domains, "S" for sets. "e" stands for expressions with "ze" for integer valued expressions. "f" stands for function expressions or defined function names, "z" for integers, and "n" for natural numbers. "$\{\}$" is for grouping, "a^+" or "$a \cdots a$" for one or more "a"s.

An abstract interpreter specification in Z consists of type definitions and the interpreter function definition. Type definitions may contain projection expressions.

[1] For a recursive definition $f = \lambda x. \cdots f \cdots$, the associated functional F_f of f is $F_f = \lambda f.\lambda x. \cdots f \cdots$.

```
f(σ:Σ, x:X̄):Ȳ                          /* applied at recursive calls of the interpreter */
begin
    if x ⋢ T_X(σ) then T_X(σ) = T_X(σ) ⊔ x;
    return T_Y(σ);
end

Tabulate(F:(Σ → X̄ → Ȳ) → Σ → X̄ → Ȳ, Σ_P: 2^Σ, x_0: X̄): void
T_X, T'_X: Σ_P → X̄;                    /* program point to pre-state */
T_Y, T'_Y: Σ_P → Ȳ;                    /* program point to post-state */
begin
    ∀σ ∈ Σ_P : T_X(σ) = ⊥_X̄,   T_Y(σ) = ⊥_Ȳ;
    T_X(σ_0) = x_0;                    /* pre-state at the program entry point */
    repeat
        ⟨T'_X, T'_Y⟩ = ⟨T_X, T_Y⟩;  /* remember the program state of the previous iteration */
        foreach σ ∈ Σ_P            /* for each program point */
            T_Y(σ) = T_Y(σ) ⊔ F(f, σ, T_X(σ));   /* evaluate and join */
    until (T_X ⊑ T'_X) ∧ (T_Y ⊑ T'_Y)            /* repeat until no movement */
end
```

Figure 2: Collecting Analysis from an Abstract Interpreter Functional F

3.1 Type Definitions

User-defined types are in two categories: set or domain. A domain is a finite-height lattice. There exist two pre-defined set types: the set of abstract syntax nodes, `syntree`, and the set of integers, `number`. When we say "two types are same" we mean that they are name-equivalent or in a synonym relation; structural equivalence does not count.

Types are defined by

(types *typedef*$^+$)

3.1.1 Sets

A set S can be defined in five ways.

(set S (elements $\iota_1 \cdots \iota_n$))	(1)
(set S (range z_1 z_2))	(2)
(set S (index ρ))	(3)
(set S (* $S_1 \cdots S_n$))	(4)
(set S' (synonym S))	(5)

(1) enumerated set	$S = \{\iota_1, \ldots, \iota_n\}$
(2) integer range set	$S = \{ i \in \mathbf{Z} \mid z_1 \leq i \leq z_2 \}$
(3) index set bound by $\rho()$	$S = \{ i \in \mathbf{Z} \mid 0 \leq i \leq \rho() \}$
(4) product set	$S = S_1 \times \cdots \times S_n$
(5) an equivalent set except in name	

All set types must be named uniquely. The element names of enumerated sets must be distinct. A `range` set's z_1 must be less than or equal to z_2. An `index` set definition is used for an index set $\{0, 1, \ldots\}$ whose bound is determined by an input program. ρ must be a name of a C procedure which returns a non-negative integer. For example, $\rho()$ may return the number of identifiers in a program.

The `synonym` set definition is used when S' and S are to be components of a product set. Since Z uses the set name as the component selector, Z requires that set component names be distinct. For example, in order to define a product set $P = S \times S$, we must introduce a synonym set S' of S, and write $P = S \times S'$.

3.1.2 Domains

A domain is a finite-height lattice. A domain can be defined in five ways.

(domain D (flat S))	(1)
(domain D (2^ S))	(2)
(domain D (* $D_1 \cdots D_n$))	(3)
(domain D (-> D_1 D_2))	(4)
(domain D' (synonym D))	(5)

(1) lifted domain of a set	$D = S_\perp^\top$
(2) powerset domain of a set	$D = 2^S$
(3) product domain	$D = D_1 \times \cdots \times D_n$
(4) atomic function domain	$D = D_1 \twoheadrightarrow D_2$
(5) equivalent domain except in name	

All domains are named uniquely. The use of the `synonym` domain definition is the same as for sets. A domain or a set cannot be defined recursively; this forces the domains to have finite height.

An element immediately above the bottom element is called an *atom*. Note that all atoms are incomparable. Henceforth, $atoms(D)$ will be the set of atoms of a lattice D.

The Partial Order and Join Operation: For a flat domain $S_\perp^\top$, all atoms $\in S$ are incomparable. For a powerset domain, the partial order is set inclusion and the join operation is set union. For a product domain $A \times B$, the partial order and join operation are component-wise: $\langle a, b\rangle \sqsubseteq \langle a', b'\rangle$ iff $a \sqsubseteq a' \wedge b \sqsubseteq b'$, and similarly for $\sqcup$. For a function domain $A \twoheadrightarrow B$, the partial order and join operation are point-wise: $f \sqsubseteq f'$ iff $f(x) \sqsubseteq f'(x)$ for all $x \in A$, and similarly for $\sqcup$.

Atomic Function Domain: An atomic function domain $D_1 \twoheadrightarrow D_2$ is a strict and distributive function domain where D_1 is an atomic domain. An atomic domain is one such that every non-bottom element x is the join of a representation set $rep(x)$ of atoms. The $rep(x)$ is defined to be the maximal such set. For example, $S_\perp^\top$ is an atomic domain with $rep(\top) = S$ and $rep(\text{atom } x) = \{x\}$.

Note that, by distributivity, every element f of an atomic function domain $D_1 \twoheadrightarrow D_2$ satisfies

$$f(x) = \sqcup_{x_i \in rep(x)} f(x_i) \qquad \forall x \in D_1.$$

Therefore, every element of an atomic function domain $D_1 \twoheadrightarrow D_2$ can be represented as a function from $atoms(D_1)$ to D_2. Z represents functions in $D_1 \twoheadrightarrow D_2$ as functions in $atoms(D_1) \rightarrow D_2$, and the user treats the functions as though in $atoms(D_1) \rightarrow D_2$ also.

Example 1 For example, let $f \in D_1 \twoheadrightarrow D_2$ and let $D_1 = 2^S$ or $S_\perp^\top$ with $S = \{a, b\}$. Z uses an element of $S \rightarrow D_2$ to represent a function in $D_1 \twoheadrightarrow D_2$. Probing f's value at $x \in D_1$ is expressed as

$$\sqcup_{x_i \in rep(x)} f(x_i).$$

Thus, instead of $f(\{a, b\})$, we write $f(a) \sqcup f(b)$. Updating f's value at $x \in D_1$ by $y \in D_2$ can be expressed, among several possibilities, as

$$f[f(x_i) \sqcup y/x_i]_{x_i \in rep(x)} \quad \text{or} \quad f[y/x_i]_{x_i \in rep(x)},$$

depending on the semantics of the operations being modeled. Thus, instead of $f[y/\{a, b\}]$, we write $f[y/a][y/b]$ or $f[f(a) \sqcup y/a][f(b) \sqcup y/b]$. □

The definition of the atomic function domain implies the following:

Property 1 *The atomic function domain $D_1 \twoheadrightarrow D_2$ is isomorphic to the product domain D_2^α where α is the number of atoms $|atoms(D_1)|$ of D_1.*

Element Height and Depth: Domain element height and depth are used in projection expressions. We define the height (respectively depth) of an element x to be the length of a maximal chain from bottom (respectively top) to x. A chain between two elements is maximal when it cannot be extended to include any more elements. In general, there exists more than one maximal chain between two elements. One immediate question is: are their lengths all same? The answer is yes, for every domain in Z; thus our definition of an element height/depth is not ambiguous.

Property 2 *For each domain in Z, every maximal chain between any two elements is of the same length.*

Proof. This property (Jordan-Hölder theorem [16]) follows when all the constructed lattices are *modular* and finite. We know that all of our lattices are of finite cardinality. All we need to show is that every constructed lattice is modular, which we can prove inductively. □

Property 3 *For each domain in Z, an element's height (respectively depth) can be defined inductively on the domain structure.*

Proof. The height of bottom $h(\perp)$ is zero. Every atom's height is 1. For a flat lattice ($S_\perp^\top$), $h(\top)$ is 2. For other cases, we can easily compute an element's height from its domain structure. Since every maximal chain from $\perp$ to x is of the same length, we can pick any maximal chain and compute its length, which is the height $h(x)$.

For $x = \{x_1, \ldots, x_n\} \in 2^S$, $h(x) = |x|$ since a maximal chain from $\perp$ to x is

$$\emptyset \sqsubseteq \{x_1\} \sqsubseteq \{x_1, x_2\} \sqsubseteq \ldots \sqsubseteq \{x_1, \ldots, x_n\}$$

whose length is $|x|$.

For $x = \langle x_1, x_2 \rangle \in D_1 \times D_2$, $h(x) = h(x_1) + h(x_2)$ since a maximal chain from $\perp$ to x is, with maximal chains $\{a_i\}$ of length n for x_1 and $\{b_j\}$ of length m for x_2,

$$\perp \sqsubseteq \langle a_1, \perp \rangle \sqsubseteq \cdots \sqsubseteq \langle a_n, \perp \rangle \sqsubseteq \langle a_n, b_1 \rangle \sqsubseteq \cdots \sqsubseteq \langle a_n, b_m \rangle$$

whose length is $n + m$.

For an atomic function $f \in D_1 \twoheadrightarrow D_2$, $h(f) = h(f(x_1)) + \cdots + h(f(x_n))$ with $\{x_i\} = atoms(D_1)$, since $D_1 \twoheadrightarrow D_2$ is isomorphic to D_2^α with $\alpha = |atoms(D_1)|$ (Property 1). □

The above property is important, because using it we can easily compute the heights (respectively depths) of domain elements for use in projections.

3.2 Projections

As we mentioned above, Z provides a simple mechanism to tune an analysis: projections of domains.

Definition 1 A map $\theta: D \rightarrow D$ is a projection if θ is monotonic, idempotent ($\theta = \theta \circ \theta$), and $Id \sqsubseteq \theta$.

Projections allow us to tune an analysis without changing any semantic operations used in defining the abstract interpreter. We can express projections independently of the semantic definition. We design "large", or maximal abstract domains and define the abstract interpreter function over these domains. After that, when we want to trim the domain structure, we define projections. Z arranges that the semantics operate over the projected domain space.

How can we compute an analysis over the projected domains? Suppose we have projections θ_A and θ_B for abstract domains A and B. For each semantic operation $f: A \rightarrow B$, we wrap f with those projections: $\theta_B \circ f \circ \theta_A$. This is correct: $f \sqsubseteq \theta_B \circ f \circ \theta_A$, because all functions are monotonic and projections are at least as safe as the identity function ($Id \sqsubseteq \theta$). Because of the idempotency of projections, it's enough to apply each projection only once (per application of f). In reality, Z uses a more sophisticated method that insures that only elements in the projected domain occur, by projecting at the moment that domain elements are created.

Example 2 Let D be a powerset domain $D = 2^S$. We can project it into several simplified domains. Suppose we regard elements whose cardinality is larger than 1 as being useless for an analysis. Then we may use a simplified domain of D in which all elements above the singleton set elements are projected to the top element. This projection θ is

$$\theta = \lambda x. \begin{cases} \top, & \text{if } h(x) = |x| > 1; \\ x, & \text{otherwise.} \end{cases}$$

□

Example 3 Suppose we have an abstract domain $D = D_1 \times D_2$. Suppose we find all the pairs $\langle \top_{D_1}, \text{"any"} \rangle$ are no more useful than $\top = \langle \top_{D_1}, \top_{D_2} \rangle$. Then, we can use a projection θ to avoid all such elements during analysis:

$$\theta = \lambda \langle d_1, d_2 \rangle. \begin{cases} \top, & \text{if } d_1 = \top_{D_1}; \\ \langle d_1, d_2 \rangle, & \text{otherwise.} \end{cases}$$

□

Z provides a limited set of projections: all projections project to the top element. A projection is expressed as

$$\texttt{(project}\ \ D\ \ pcond\texttt{)}$$

All elements of D that satisfies the condition $pcond$ are projected to $\top_D$. Note that this operation is qualified as a projection (see Definition 1). Abstract syntax of a projection condition is as follows.

$$
\begin{array}{lcll}
pcond & ::= & \texttt{top?} & (1)\\
 & | & \texttt{(height>}\ n\texttt{)} & (2)\\
 & | & \texttt{(depth<}\ n\texttt{)} & (3)\\
 & | & \texttt{(*-and}\ pcond_1 \cdots pcond_n\texttt{)} & (4)\\
 & | & \texttt{(*-or}\ pcond_1 \cdots pcond_n\texttt{)} & (5)\\
 & | & \texttt{(->all}\ pcond\texttt{)} & (6)\\
 & | & \texttt{(->exists}\ pcond\texttt{)} & (7)
\end{array}
$$

Each projection condition represents a predicate as follows.

(1) $x = \top$?
(2) height$(x) > n$?
(3) depth$(x) < n$? (dual of height)
(4) $x \in D_1 \times \cdots \times D_n . \forall i \in \{1, \ldots, n\} : pcond_i(x \downarrow i)$
(5) $x \in D_1 \times \cdots \times D_n . \exists i \in \{1, \ldots, n\} : pcond_i(x \downarrow i)$
(6) $x \in D_1 \twoheadrightarrow D_2 . \forall$atom $a \in D_1 : pcond(x(a))$
(7) $x \in D_1 \twoheadrightarrow D_2 . \exists$atom $a \in D_1 : pcond(x(a))$

3.3 Interpreter Functions

After types (sets and domains) are defined, the semantic definition follows. A semantics definition is a sequence of function declarations and definitions. The main interpreter function is called the *semantic function*. All other functions are called *auxiliary functions*. Only one semantic function can be defined.

The type of each function must be declared before it is used:

$$\texttt{(}f\ \ \texttt{(from}\ \tau_1 \cdots \tau_n\texttt{)}\ \ \texttt{(to}\ \tau\texttt{))}$$

The semantic function is defined with a reserved word `semantics`:

$$\texttt{(semantics}\ \ f\ \ \texttt{(}x_1 \cdots x_n\texttt{)}\ \ e\texttt{)}$$

The first argument x_1's type should always be `syntree`. A `start` declaration (optional)

$$\texttt{(start}\ \ e_2 \cdots e_n\texttt{)}$$

specifies the pre-state that holds at the program entry point. Note that a pre-state consists of the arguments of the semantics function excluding its first argument of type `syntree`. Each e_i should match with its corresponding argument type τ_i of the semantics function. Without the `start` declaration, $\perp_{\tau_i}$'s are assumed for the starting pre-state.

Auxiliary function definitions are of three kinds:

$$
\begin{array}{ll}
\texttt{(function}\ \ f\ \ \texttt{(}x_1 \cdots x_n\texttt{)}\ \ e\texttt{)} & (1)\\
\texttt{(mfunction}\ \ f\ \ \texttt{(}x_1 \cdots x_n\texttt{)}\ \ e\texttt{)} & (2)\\
\texttt{(tfunction}\ \ f\ \ \texttt{(from}\ S_1 \cdots S_n\texttt{)}\ \ \texttt{(to}\ \tau\texttt{)} & \\
\qquad\qquad\qquad \{\texttt{(}e_1 \cdots e_n\ \ e\texttt{)}\}^{+}\texttt{)} & (3)
\end{array}
$$

(1) `function` specifies a usual function. Given an input pair, evaluate the function body and returns its result.

(2) `mfunction` specifies a memoized function. All ever-computed pairs of input and output are recorded in order to bypass the computation whenever the function is applied to a previous input set. (Garbage collection is used to recover space in these tables periodically.)

(3) `tfunction` specifies a tabularized function. Before the analysis, the whole graph of the function is recorded in an array. Every function call is an array access to the corresponding entry. Z requires that the argument types of a tabularized function be sets whose size can be determined independent of the input program.

Abstract syntax of Z for semantics expression part is shown in Figure 3. Every semantics expression denotes a value of a unique type τ. A type is a user-defined type, a pre-defined type (`syntree` or `number`), or one described by a function expression. Integers (of type `number`) are also used as boolean values: 0 for false, others for true.

User-defined types and the pre-defined types constitute the *basic types*. The type of every function expression is from a number of basic types to a basic type. For each basic type, Z provides a set of constants and primitive operations over its values. Certainly, the expression constructs shown in Figure 3 are not defined for all types of values. The reader may want to see [25] for their formal semantics and type rules.

3.4 Defining the Abstract Syntax of a Target Language

As might be noted, Z does not provide any construct to specify the abstract syntax of a target language. System Z does not generate the syntax tree, nor does it provide any operator to access its nodes. The user must write C procedures for those operations and declare them inside Z. Declaring those C procedures in Z is done in the same way as for other Z functions. Because no foreign types are allowed in Z, two reserved type names, `syntree` and `number`, may be used to declare the types of those procedures. The correspondence between C types and these two Z types must be defined in a C header file (`user.h`). As an example, suppose the user has a C procedure `isPlus`, which returns true when it's argument is a plus node. Inside Z, the user declares it, before its use, as "`(isPlus (from syntree) (to number))`". Inside `user.h`, `syntree` and `number` are defined as, for example, "`typedef struct node* syntree`" and "`typedef int number`".

Currently, we have implemented ANSI C, FORTRAN, and SCHEME front-ends for an intermediate language MIL (MIPRAC [11, 10, 9] Intermediate Language), along with a set of routines to access MIL programs, so that Z may be used to analyze programs in these source languages.

4 Experiments

In this section, we present examples to demonstrate system Z's capabilities, focusing on the projection facility. We first present cost and accuracy variations depending on projections. Even though it is usually true that projections reduce the analysis cost at the expense of accuracy, there are some cases in which they reduce only the cost without any change in accuracy. We present one such case below.

We have designed two analyses in Z: constant propagation analysis and alias analysis. The target language is MIL [11, 10]. Since we have parsers from ANSI C and FORTRAN into MIL, we can use C and FORTRAN programs as test suites. The two analyses are embodied in a single abstract interpreter definition. The interpreter is defined over a couple of expensive domain structures (e.g., the

τ	::=		$D \mid S \mid$ **number** $\mid$ **syntree**	types
f	::=		(lambda (from $\tau_1 \cdots \tau_n$) (to τ) ($x_1 \cdots x_n$) e)	
		$\mid$	*defined function name*	
e	::=	$\mid$	ι	enumerated set element
		$\mid$	x	bound name
		$\mid$	(_ x ze)	indexed bound name x_{ze}
		$\mid$	(τ bottom)	$\perp_\tau$
		$\mid$	(τ top)	$\top_\tau$
		$\mid$	(τ e)	type cast
		$\mid$	(τ $e_1 \cdots e_n$)	powerset elmt $\{e_1, \cdots, e_n\}_\tau$ or product set/lattice elmt $\langle e_1, \cdots, e_n \rangle_\tau$
		$\mid$	(τ (each x e))	ftn lattice elmt $[\lambda x.e]_\tau$
		$\mid$	(! e $n_1 \cdots n_k$)	$e \downarrow n_1 \cdots \downarrow n_k$
		$\mid$	(! e $\tau_1 \cdots \tau_n$)	$e \downarrow \tau_1 \cdots \downarrow \tau_n$
		$\mid$	(@ e $e_1 \cdots e_n$)	$e(e_1) \cdots (e_n)$
		$\mid$	(<= e_1 e_2)	$e_1 \sqsubseteq e_2$
		$\mid$	(= e_1 e_2)	equivalence
		$\mid$	($\mid\mid$ e)	cardinality $\lvert e \rvert$ of a powerset elmt e
		$\mid$	(in e_1 e_2)	$e_1 \in e_2$
		$\mid$	(join e_1...e_n)	$e_1 \sqcup \cdots \sqcup e_n$
		$\mid$	(join (for x ze_1 ze_2 e))	$\sqcup_{ze_1 \le x \le ze_2} e$
		$\mid$	(mapjoin f e)	$\sqcup_{x \in e} f(x)$
		$\mid$	([/] e (by e_{1_1} e_{1_2})$\cdots$(by e_{n_1} e_{n_2}))	$e[e_{1_2}/e_{1_1}] \cdots [e_{n_2}/e_{n_1}]$
		$\mid$	(map[/] e f e_1)	$e[f(x)/x_\tau]_{x \in e_1}$
		$\mid$	(dom e)	$\{\, x \mid e(x) \neq \perp \,\}_\tau$
		$\mid$	(nth e ze)	ze-th elmt of a powerset elmt e
		$\mid$	(f $e_1 \cdots e_n$)	function apply $f(e_1, \cdots, e_n)$
		$\mid$	(if ze e_1 e_2)	conditional
		$\mid$	(switch $\{ze\ e\}^+$ default e)	switch to the first true branch
		$\mid$	(begin be^+ e)	sequential bindings and body
		$\mid$	error	semantics error
		$\mid$	ze	integer (also boolean) valued expression
be	::=		(let x tag e)	binding $x = e$
		$\mid$	(let (_ x i) tag (for i ze_1 ze_2 e))	indexed binding $x_i = e$ for $ze_1 \le i \le ze_2$
		$\mid$	(let (_ x z) tag e_1 (_ x i) tag (for i ze_1 ze_2 e_2))	indexed binding with initialization $x_i = e_2$ for $ze_1 \le i \le ze_2$ with $x_z = e_1$
tag	::=		:τ	
		$\mid$	$empty$	
ze	::=		x	bound name
		$\mid$	z	integer constant
		$\mid$	(and ze_1 ze_2) $\mid$ (or ze_1 ze_2) $\mid$ (not ze)	
		$\mid$	(rop ze_1 ze_2) $\mid$ (bop ze_1 ze_2)	
		$\mid$	(bop (for i ze_1 ze_2 ze_3))	$bop_{i=ze_1}^{ze_2} ze_3$
bop	::=		+ $\mid$ - $\mid$ * $\mid$ /	integer binary operation
rop	::=		= $\mid$ < $\mid$ > $\mid$ <= $\mid$ >=	integer relation

Figure 3: Abstract Syntax of Semantics Expressions in Z

abstract integer is a powerset domain of an integer range set), which is trimmed later by series of projection expressions. The definition has 667 lines. Due to space limitation we show only parts of the definition in Appendix A.1. (The reader may want to see [25] for the complete definition.) The generated C code for the analysis has 12K lines. Its executable binary size is 463K bytes.

4.1 Cost Variations Depending on Projections

Tables in Figure 4 show the cost variations depending on the projections. The generated analyzers are compiled using `gcc` and are executed on a SPARCstation 2. The four test programs are `wator.c` (C), which simulates a simple ecological system, `gauss.f` (FORTRAN), which is a Gaussian elimination program, `amoeba.f`, which optimizes a nonlinear function of multiple variables, and `simplex.f`, which optimizes a system of linear constraints. Sizes of the programs after translated into MIL are:

`wator.c`: 45 procedures, 3467 exprs, 166 vars.
`gauss.f`: 54 procedures, 4710 exprs, 172 vars.
`amoeba.f`: 36 procedures, 6062 exprs, 221 vars.
`simplex.f`: 44 procedures, 8739 exprs, 250 vars.

We tried three projections:

- `(project Z (height> 1))`: domain `Z` is the abstract integer defined as a powerset domain of an integer range set. By this projection, `Z`'s elements which are higher than the singleton sets are projected to top. This is the domain used in conventional constant propagation analysis.
- `(project Z (height> 2))`: `Z`'s elements higher than the two-element sets are projected to top.
- `(project Gs (height> 0))`: domain `Gs` is used to differentiate instances of local variables. The projection collapses all the abstract instances into a single one. An instance of a variable is a memory portion allocated for the variable by a new activation of its owner procedure (which has the variable as a parameter or a local variable). The projection specifies that there be only one abstract instance.

Memory consumption and the amount of garbage reclaimed are the average amounts (in mega bytes) after a garbage collection. As expected, the analysis cost decreases as we project the domains into simpler ones. The shortened heights of the projected domains reduce the number of iterations required to reach a fixpoint. Note that the times are on the order of several minutes. Currently, system Z does not perform any optimization of the input specification. The system might include a phase to analyze the input specification and manipulate it into a simpler one.

While projections reduce the cost of analysis, they decrease its accuracy. This is because projections map a domain element into a higher one (in our case, into top) which is a less accurate value. In the next section we illustrate how projections decrease the accuracy of the analysis.

4.2 Accuracy Variations Depending on Projections

We will show, for an example program, how a projection expression changes the analysis in accuracy. Let us consider the left-hand side C program in Figure 5. It is written to have several aliases and interesting constant properties. Note that the variable `n` can have only two values: 0 or 1. Note that the formal arguments `x` and `y` of the procedure `g` points to `i` or `j`.

Firstly, we project the abstract integer domain `Z` using

```
(project Z (height> 1)).
```

This projection expression makes the powerset domain `(domain Z (2^ setZ))` becomes equivalent to the flat domain `(domain Z (flat setZ))` used in conventional constant propagation analysis. The analysis result at the program end point is in the left hand side of Figure 6. We use the output of the display routines generated by the system. The format rule is simple. The first column has element's domain name. A function domain element f is printed as `[`x_0 `->` $f(x_0)$`]`$\cdots$`[`x_n `->`$f(x_n)$`]` for defined entries $\{x_0,\ldots,x_n\}$. A powerset domain element is printed in enumerated set notation. A product domain element x is printed as `<`$x \downarrow 0$`,...,`$x \downarrow n$`>`. The bottom (respectively top) element is printed as _(respectively ^).

Note that the value of `n` ("`Id 8->`") is the top integer, and each value of `x` ("`Id 12->`") and `y` ("`Id 13->`") is a location of `i` or `j` ("`{5,6}`").

Next, we project the `Z` domain using

```
(project Z (height> 2)).
```

This projection expression gives a richer structure than the previous one. Between singleton sets and the top value exist two-element sets. When a two-element set is obtained we can retain to it instead of projecting it to top as in the previous case. The result is the right hand side of Figure 6. The analysis output becomes sharper. It can detect that `n` is a two-valued ("`{0,1}`") variable instead of an all-valued ($\top_{\mathtt{Z}}$) variable. This result also sharpens the alias information, that `x` and `y` are pointing to only one location: `x` to `i` ("`{5}`") and `y` to `j` ("`{6}`").

4.3 Projection Without Loss of Accuracy

As second example, we consider an analysis to find procedures whose arguments are all constants. When a procedure is called always with the same actual parameters we can remove code for the parameter passing. This example is a case in which a projection expression reduces the analysis cost without a loss of accuracy. Due to space limitation we show only parts of the definition in Appendix A.2. (The reader may want to see [25] for the complete definition.) The definition has 696 lines.

Domain `Q` records, for each procedure, abstract integer values bound to its arguments:

```
(domain Q (-> Proc Arg))   (1)
(domain Arg (-> Id Z))     (2)
```

(1) maps each procedure $\in$ `Proc` to a value $\in$ `Arg`
(2) maps each parameter $\in$ `Id` to a value $\in$ `Z`

Suppose that whenever one of a procedure's arguments is bound to distinct values, we wish not to consider the other arguments. For example, such a procedure might be disqualified for the planned optimization; if we suppose that having only one parameter of a procedure bound to varying values is as useless to us as having every parameter bound to changing values. Thus, we project every element of `Arg` whose entry contains the $\top_{\mathtt{Z}}$ into the $\top_{\mathtt{Arg}}$:

```
(project Arg (->exists top?)).
```

This projection will not affect our optimization but will reduce the analysis cost.

Cost Variations (`wator.c`: 45 procedures, 3467 exprs, 166 vars in MIL)				
Projections	Iterations	CPU sec	MemUse(Garbage)(M)	♯GC
h(`Z`) > 1	340,978	529.21	6.14 (5.27)	94
h(`Z`) > 1 h(`Gs`) > 0	157,678	112.76	7.01 (5.19)	4
h(`Z`) > 2 h(`Gs`) > 0	291,349	241.75	6.12 (5.24)	18

Cost Variations (`gauss.f`: 54 procedures, 4710 exprs, 172 vars in MIL)				
Projections	Iterations	CPU sec	MemUse(Garbage)(M)	♯GC
h(`Z`) > 1	376,660	782.29	6.22 (5.29)	186
h(`Z`) > 1 h(`Gs`) > 0	248,721	258.78	6.11 (5.28)	18
h(`Z`) > 2 h(`Gs`) > 0	271,922	283.74	6.01 (5.28)	21

Cost Variations (`amoeba.f`: 36 procedures, 6062 exprs, 221 vars in MIL)				
Projections	Iterations	CPU sec	MemUse(Garbage)(M)	♯GC
h(`Z`) > 1	784,270	3,647.19	6.72 (5.34)	1,115
h(`Z`) > 1 h(`Gs`) > 0	541,199	519.55	6.42 (5.32)	57
h(`Z`) > 2 h(`Gs`) > 0	572,296	530.38	6.36 (5.30)	53

Cost Variations (`simplex.f`: 44 procedures, 8739 exprs, 250 vars in MIL)				
Projections	Iterations	CPU sec	MemUse(Garbage)(M)	♯GC
h(`Z`) > 1	1,084,662	6,134.81	7.01 (5.35)	1,998
h(`Z`) > 1 h(`Gs`) > 0	751,400	969.17	6.42 (5.31)	84
h(`Z`) > 2 h(`Gs`) > 0	779,247	997.14	6.47 (5.32)	93

Figure 4: Analysis Cost Variations Depending on Projections

Consider the program in the right hand side of Figure 5. Its analysis outputs (with/without the projection) at the program end point are shown in Figure 7 (the right hand side is with the projection). Both analysis results will have same effect on our optimization: both detect that function `h` is called with constant arguments and the others with varying values.

5 Conclusion and Future Work

In this paper we have presented *system Z* which automates the generation and management of interprocedural program analyses. The system provides a high-level facility (projection expressions) by which the user can tune the analyses in accuracy and cost. We demonstrated the system's capabilities by experiments with a set of generated analyzers which can analyze ANSI C, FORTRAN, and SCHEME programs. We show how we can use the projections to change an analysis in accuracy and cost. We have designed the specification language to be general enough to specify a large class of global analyses for a wide range of target languages, and to be abstract enough to specify an analysis succinctly. Whether this claim is substantiated remains to be seen as we try other analyses than those presented in this paper; we plan to expand the language if necessary. The system provides the run-time supports (e.g. garbage collection) and other extra routines (e.g. data type pretty printers, dump/restore routines for the computed fixpoint) needed to debug an abstract interpreter specification and to examine the analysis result.

We can summarize this paper's contributions as follows:

- By System Z, we showed that many aspects of the generation and management of interprocedural program analyses can be automated. It is possible to generate an analysis from a high-level abstract interpreter definition, and to tune it using projection expressions over the types (value spaces) of the interpreter.
- We presented a simple high-level specification language in which we can specify detailed semantic analyses for languages like C which is considered hard to analyze. System Z is to be used as a fast prototyping tool for such analyses.
- System Z can facilitate the use of abstract interpretation in designing an analysis for imperative as well as functional programming languages.

There are several extensions worth trying.

- More general projection expressions. Currently, all projections project to the top element. This set of projections is shown to be useful, but more elaborate projection expressions are needed for some cases. For example, we may want to project a domain D such that the elements below an $x \in D$ are projected to x and those above x to top.
- Theorem generator. Here, a program analysis is a data collection procedure. As a next step after the collection, we usually need an analysis of the result: we must derive information which summarizes the collected data in a suitable form for an intended optimization. The system might provide a specification language in which the user can write rules for data analysis.

```
/*
 * Constant Propagation and Alias Analysis
 * Demo Input
 */
void f(int *x) {
  if (*x <= 0) *x = 1;
  else *x = 0;
}

void g(int *x, int *y) {
  *x = 1;
  *y = 2;
}

#define BOUND 100000
void main () {
  int i, j, k, n;
  int *a, *b;

  i = j = n = 0;
  /*
   * For constant propagation analysis.
   * Note the value of n across iterations.
   */
  for (k=0; k<BOUND; k++)
    f(&n);
  /*
   * For alias analysis.
   * Note the parameter bindings of g.
   */
  a = &i;
  b = &j;
  if (n >= 0) g(a,b);
  else g(b,a);
}
```

```
/*
 * Constant Call Analysis Demo Input
 */

void f(int x, int y, int z) {
  if (x < y) h(y, z);
  else h(z-10,y+10);
}

void h(int l, int m) {
  int i, sum = 0;
  for (i=1; i<m; i++) sum++;
}

void g(int p, int q) {
  int i, prod = 1;
  for (i=p; i<q; i++) prod *= i;
}

#define BOUND 100000
void main() {
  int i;
  /*
   * First loop calls f, which calls h.
   * Note that h's params are all constants.
   * Second loop calls g or h.
   * Still, h's params are all constants.
   * All others have varying parameters.
   */
  for (i=0; i<BOUND; i++) {
    f(i, 10, 20);
  }
  for (i=0; i<BOUND; i++) {
    if (i < 100) g(i,i+i);
    else h(10,20);
  }
}
```

Figure 5: Input C programs (expository purposes)

- User specified fixpoint computation optimization. We have used a general fixpoint computation method. Depending on target languages, however, we can improve its performance. For example, for a language which enables us to determine, before the analysis starts, the control flow graph of an input program, the fixpoint computation method can be tailored for the control flow graph. As in [21], system Z can provide a facility by which the user can specify such optimizations when possible.
- Optimization of the generated analysis. There are ways in which we can further improve the performance of the generated C code. For example, currently domain elements are manipulated through the limited set of interface procedures of the domain module. The generated analysis could be made more efficient, by allowing it to manipulate the representations of domain elements more directly. The system might also include a phase to analyze the input specification and manipulate it into a simpler one.

References

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[2] Jeffrey M. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 119–131, January 1977.

[3] Geoffrey L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Department of Computing, Imperial College, University of London, March 1987.

[4] Geoffrey L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. The MIT Press, 1991.

[5] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–56, 1988.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[7] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[8] Harald Ganzinger and Robert Giegerich. A truly generative semantics-directed compiler generator. In *Proceedings of SIGPLAN '82 Symposium on Compiler Construction*, volume 17 of *SIGPLAN Notices*, pages 172–184, 1982.

[9] Williams Ludwell Harrison III. Generalized iteration space and the parallelization of symbolic programs (extended abstract). In *Proceedings of the workshop on compilation of (symbolic) languages for parallel computers*, October 1991.

```
                                     [                         |       [
                   n --->>      Id  8->                        |  Id  8->            <<--- n
                                SS                             |  SS
                                     [                         |       [
                                G     D->                      |  G     D->
                                V     <                        |  V     <
                                L      _,                      |  L      _,
                                C      _,                      |  C      _,
          value of n --->>      Z      ^>]                     |  Z      {0,1}>]]   <<--- value of n

                                     [                         |       [
              x of g --->>      Id  12->                       |  Id  12->          <<--- x of g
                                SS                             |  SS
                                     [                         |       [
                                G     D->                      |  G     D->
                                V     <                        |  V     <
                                L      <                       |  L      <
x is alias to i(5) or j(6)--->> Ids     {5,6},                 |  Ids     {5},      <<--- x is alias to i (5)
                                P                              |  P
                                          [                    |            [
                                Proc      2->                  |  Proc      2->
                                Gs         {D}]>,              |  Gs         {D}]>,
                                C      _,                      |  C      _,
                                Z      {0}>]]                  |  Z      {0}>]]

                                     [                         |       [
              y of g --->>      Id  13->                       |  Id  13->          <<--- y of g
                                SS                             |  SS
                                     [                         |       [
                                G     D->                      |  G     D->
                                V     <                        |  V     <
                                L      <                       |  L      <
y is alias to i(5) or j(6)--->> Ids     {5,6},                 |  Ids     {6},      <<--- y is alias to j (6)
                                P                              |  P
                                          [                    |            [
                                Proc      2->                  |  Proc      2->
                                Gs         {D}]>,              |  Gs         {D}]>,
                                C      _,                      |  C      _,
                                Z      {0}>]]                  |  Z      {0}>]]
```

Figure 6: Accuracy Variations Depending on Projections

```
                                       |  Q
                                       |     [
                                       |  Proc  6->        <<--- g
                                       |  Arg
                                       |       [
                                       |  Id    9->        <<--- param p
                                       |  Z     ^]
                                       |       [
                                       |  Id    10->       <<--- param q
                           Q           |  Z     ^]]
                              [        |     [
              g   --->>    Proc  6->   |  Proc  7->        <<--- h
                           Arg ^]      |  Arg
                              [        |       [
              h   --->>    Proc  7->   |  Id    14->       <<--- param l
                           Arg         |  Z     {10}]
                                [      |       [
        param l --->>      Id    14->  |  Id    15->       <<--- param m
                           Z     {10}] |  Z     {20}]]
                                [      |     [
        param m --->>      Id    15->  |  Proc  8->        <<--- f
                           Z     {20}]]|  Arg
                              [        |       [
              f  --->>     Proc  8->   |  Id    19->       <<--- param x
                           Arg ^]      |  Z     ^]
                                       |       [
                                       |  Id    20->       <<--- param y
                                       |  Z     {10}]
                                       |       [
                                       |  Id    21->       <<--- param z
                                       |  Z     {20}]]
```

Figure 7: Accuracy Invariation for Some Projections

[10] Williams Ludwell Harrison III. *Semantic Analysis of Symbolic Programs for Automatic Parallelization*. Book in preparation, 1992.

[11] Williams Ludwell Harrison III and Zahira Ammarguellat. A program's eye view of miprac. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing (Also as a UIUC CSRD Report 1227)*. MIT Press, August 1992.

[12] Matthew S. Hecht. *Flow Analysis of Comptuer Programs*. Elsevier North-Holland, Inc., 1977.

[13] Paul Hudak and Jonathan Young. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13(2):269–290, April 1991.

[14] Kai Koskimies, Otto Nurmi, Jukka Paakki, and Seppo Sippu. The design of a language processor generator. *Software-Practice and Experience*, 18(2):107–135, 1988.

[15] Eugene W. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[16] D.E. Rutherford. *Introduction to Lattice Theory*. Hafner Publishing Company, New York, 1965.

[17] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[18] Masataka Sassa. Rie and jun: Towards the generation of all compiler phases. In *Lecture Notes in Computer Science*, volume 477, pages 56–70. 1990.

[19] Bernahard Steffen. Data flow analysis as model checking. In *Lecture Notes in Computer Science*, volume 526, pages 346–364. 1991.

[20] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[21] Steven W. K. Tjiang and John L. Hennessy. Sharlit - a tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 82–93, June 1992.

[22] G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1989.

[23] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, 1980.

[24] Reinhard Wilhelm. Global flow analysis and optimization in the mug2 compiler generating system. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 5. Prentice-Hall, 1981.

[25] Kwangkeun Yi and Williams Ludwell Harrison III. Z: Interprocedural data flow analysis specification language (in preparation). Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1992.

[26] Jonathan H. Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages*. PhD thesis, Yale University, May 1989.

A Abstract Interpreter Definitions in Z

The abstract interpreter definitions which have been used for the experiments in Section 4 are presented in this appendix. We include the type definitions and only the procedure call interpretation part. The target language is MIL [11, 10].

A.1 Parts of Constant Propagation and Alias Analysis Specification

Type definitions:

```
(types
 (domain PRE (* E S P))       ; pre state
 (domain POST (* S V))        ; post state
 (domain S (-> Id SS))        ; store = identifier -> substore
 (domain SS (-> Gs V))        ; substore = instance -> value
 (domain V (* L C Z))         ; value = location X closure X integer
 (domain L (* Ids P))         ; location = identifiers X instance-indicator
 (domain C (* Procs E))       ; closure = procedure-indices X environment
 (domain P (-> Proc Gs))      ; instance-indicator = procedure-index -> movements
 (domain E (synonym P))       ; E = P
 (domain Ids (2^ setId))      ; powerset identifier
 (domain Id (flat setId))     ; flat identifier
 (domain Z (2^ setZ))         ; integer                  (project Z (height> 1))
 (domain T (flat setB))       ; boolean
 (domain Procs (2^ setProc))  ; powerset procedure index
 (domain Proc (flat setProc)) ; flat procedure index
 (domain Gs (2^ G))           ; procedure string         (project Gs (height> 0))
 (set G (elements EPS D DD U UU UD))     ; set of procedure strings
 (set setId (index NumOfPlaces))         ; set of identifier indices
 (set setProc (index NumOfProcs))        ; set of procedure indices
 (set setB (elements True False))        ; set of boolean values
 (set setZ (range -999999 999999))       ; set of finite range integers
 (set movement (elements UP DOWN))       ; set of procedural movements (call, return)
)
```

Declaring the functions to access source program's astract syntax tree: (the user must define these functions in C)

```
(NumOfSub (from syntree) (to number))
(NumOfParam (from setProc) (to number))
(NumOfLocal (from setProc) (to number))
(NthSubexpr (from syntree number) (to syntree))
(NthParam NthLocal (from setProc number) (to setId))
(Body (from setProc) (to syntree))
(isProc isClosure (from syntree) (to number))
```

The interpretation function: (we include only the procedure call part.)

```
;;;; The semantic function
(Eval (from syntree E S P) (to POST))    (start (zeroP) (S bottom) (zeroP))
(semantics Eval (x e s p)
  (switch
   (or (isCall x) (isApply x))
   (begin
    (let n (NumOfSub x))
    (let num_actual (- n 1))
    ;;;; evaluate the parameters
    (let (_post 1) (Eval (NthSubexpr x 1) e s p)
         (_post i) (for i 2 n
                        (Eval (NthSubexpr x i)
                              e
                              (! (_post (- i 1)) S)
                              p)))
    (let (_v i) (for i 1 n (! (_post i) V))) ; closure and actual params
    (let sl (! (_post n) S))
    (let c (! (_v 1) C))    ; closure value = <procs, env>
    (let procs (! c 1))     ; procedure indices
    (let f (! c 2))         ; environment
    ;;;; apply procedures
    (mapjoin
     (lambda (from setProc) (to POST) (alpha)
       (begin
        (let mom (Body alpha))           ; procedure body
        (let p1 (moveP p alpha DOWN))    ; record procedural movement
        (let env1 (moveP f alpha DOWN))  ; record procedural movement in environment
        (let s1 (moveS sl alpha DOWN))   ; record procedural movement in store
        (let num_param (NumOfParam alpha))
        (if (< num_actual  num_param)
            (POST bottom)
          (begin
           (let params :Ids (join (for i 1 num_param
                                  (Ids (NthParam alpha i)))))
           (let (_s2 0) s1       ;;;; bind parameters
                (_s2 i) (for i 1 num_param
                             (begin
                              (let param :setId (nth params i))
                              (let ss (slashSS (@ (_s2 (- i 1)) (Id param))
                                               (_v (+ i 1))
                                               (@ env1 (Proc alpha))))
                              ([/] (_s2 (- i 1)) (by (Id param) ss)))))
           (let num_local (NumOfLocal alpha))
           (let locals :Ids (join (for i 1 num_local
                                  (Ids (NthLocal alpha i)))))
           (let (_s3 0) (_s2 num_param)    ;;;; initialize local variables
                (_s3 i) (for i 1 num_local
                             (begin
                              (let local :setId (nth locals i))
                              (let ss (slashSS
                                       (@ (_s3 (- i 1)) (Id local))
                                       (V (L bottom)
                                          (C bottom)
                                          (Z (setZ 0)))
                                       (@ env1 (Proc alpha))))
                              ([/] (_s3 (- i 1)) (by (Id local) ss)))))
           (Eval mom env1 (_s3 num_local) p1)   ;;;; evaluate the procedure body
           ))))
     procs
     ))
  ))
```

A.2 Parts of Constant-Call Analysis Specification

Type definitions:

```
(types
 (domain PRE (* E S P Q))       ; pre state
 (domain POST (* S V Q))        ; post state
 (domain S (-> Id SS))          ; store = identifier -> substore
 (domain SS (-> Gs V))          ; substore = instance -> value
 (domain V (* L C Z))           ; value = location X closure X integer
 (domain L (* Ids P))           ; location = identifiers X instance-indicator
 (domain C (* Procs E))         ; closure = procedure-indices X environment
 (domain P (-> Proc Gs))        ; instance-indicator = procedure-index -> movements
 (domain E (synonym P))         ; E = P
                                ; Q and Arg are for constant-call analysis
 (domain Q (-> Proc Arg))       ; Q = procedure-index -> arguments
 (domain Arg (-> Id Z))         ; arguments = identifier -> integer
                                ; (project Arg (->exists top?))
 (domain Ids (2^ setId))        ; powerset identifier
 (domain Id (flat setId))       ; flat identifier
 (domain Z (2^ setZ))           ; integer
 (domain T (flat setB))         ; boolean
 (domain Procs (2^ setProc))    ; powerset procedure index
 (domain Proc (flat setProc))   ; flat procedure index
 (domain Gs (2^ G))             ; procedure string
 (set G (elements EPS D DD U UU UD))     ; set of procedure strings
 (set setId (index NumOfPlaces))         ; set of identifier indices
 (set setProc (index NumOfProcs))        ; set of procedure indices
 (set setB (elements True False))        ; set of boolean values
 (set setZ (range -999999 999999))       ; set of finite range integers
 (set movement (elements UP DOWN))       ; set of procedural movements (call, return)
)
```

Declaring the functions to access source program's abstract syntax tree: (the user must define thses functions in C)

```
(NumOfSub (from syntree) (to number))
(NumOfParam (from setProc) (to number))
(NumOfLocal (from setProc) (to number))
(NthSubexpr (from syntree number) (to syntree))
(NthParam NthLocal (from setProc number) (to setId))
(Body (from setProc) (to syntree))
(isProc isClosure (from syntree) (to number))
```

The interpretation function:(we include only the procedure call part.)

```
;;;; The semantic function
(Eval (from syntree E S P Q) (to POST)) (start (zeroP) (S bottom) (zeroP) (Q bottom))
(semantics Eval (x e s p q)
  (switch
   (or (isCall x) (isApply x))
   (begin
     (let n (NumOfSub x)) (let num_actual (- n 1))
     ;;;; evaluate the parameters
     (let (_post 1) (Eval (NthSubexpr x 1) e s p q)
          (_post i) (for i 2 n
                         (Eval (NthSubexpr x i) e (! (_post (- i 1)) S) p
                               (! (_post (- i 1)) Q))))
     (let (_v i) (for i 1 n (! (_post i) V))) ; closure and actual params
     (let sl (! (_post n) S))
     (let q (! (_post n) Q))
     (let c (! (_v 1) C))    ; closure value = <procs, env>
     (let procs (! c 1))     ; procedure indices
     (let f (! c 2))         ; environment
     ;;;; apply procedures
     (mapjoin
      (lambda (from setProc) (to POST) (alpha)
        (begin
          (let mom (Body alpha))          ; procedure body
          (let p1 (moveP p alpha DOWN))   ; record procedural movement
          (let env1 (moveP f alpha DOWN)) ; record procedural movement in env
          (let s1 (moveS sl alpha DOWN))  ; record procedural movement in store
          (let arg1 (@ q (Proc alpha)))
          (let num_param (NumOfParam alpha))
          (if (< num_actual  num_param) (POST bottom)
            (begin (let params :Ids (join (for i 1 num_param
                                          (Ids (NthParam alpha i)))))
                   ;;;; record the arguments value in Q = Proc -> Arg
                   (let (_arg2 0) arg1
                        (_arg2 i) (for i 1 num_param
                                       ([/] (_arg2 (- i 1))
                                            (by (Id (nth params i))
                                                (ConstArg (_ v (+ i 1)))))))
                   (let q2 (join q ([/] q (by (Proc alpha) (_arg2 num_param)))))
                   (let (_s2 0) s1       ;;;; bind parameters
                        (_s2 i) (for i 1 num_param
                                     (begin
                                      (let param :setId (nth params i))
                                      (let ss (slashSS (@ (_s2 (- i 1)) (Id param))
                                                       (_v (+ i 1))
                                                       (@ env1 (Proc alpha))))
                                      ([/] (_s2 (- i 1)) (by (Id param) ss)))))
                   (let num_local (NumOfLocal alpha))
                   (let locals :Ids (join (for i 1 num_local
                                          (Ids (NthLocal alpha i)))))
                   (let (_s3 0) (_s2 num_param)    ;;;; initialize local variables
                        (_s3 i) (for i 1 num_local
                                     (begin (let local :setId (nth locals i))
                                            (let ss (slashSS
                                                     (@ (_s3 (- i 1)) (Id local))
                                                     (V (L bottom)
                                                        (C bottom)
                                                        (Z (setZ 0)))
                                                     (@ env1 (Proc alpha))))
                                            ([/] (_s3 (- i 1)) (by (Id local) ss)))))
                   (Eval mom env1 (_s3 num_local) p1 q2)  ;;;; evaluate the procedure body
          ))))
      procs
      ))
))
```