SUIF Program Analysis Using System Z2

Seong-Hoon Kim, Kwangkeun Yi, Hyun-jun Eo, Kwang-Moo Choe
Department of Computer Science
Korea Advanced Institute of Science and Technology
{kimsh,kwang,poisson,choe}@cs.kaist.ac.kr

1 Introduction

System Z2(figure 1), which is a new version of [5], is a program analyzer generator based on formal semantics of the input program. The input is a high-level specification of a data-flow analysis (an abstract interpreter[3]). The output is a C program for the specified analysis. The generated analyzer is linked with the target language (e.g., SUIF) parser, and analyzes the input programs (e.g., SUIF programs). The analysis result is a table from program points (syntax tree nodes) to abstract program states, each of which approximates the run-time states that occur at the program point.

Some merits of System Z2 come from the power of the abstract interpretation framework. In this framework, concern about differentiating the inter-procedural and intra-procedural analyses disappears, computing the control-flow graph prior to program analysis is not necessary (hence we can design an analysis for high-order languages), and proving the correctness of our analyses can be done. Thanks to this framework, Z2 can also provide a simple, high-level command set (called *projections*) by which the user can tune the cost-accuracy trade-off of the generated analyzers.

The Z2 system has been used to prototype analyzers (constant propagation[5], def-use chain[1], exception analysis[6]) for programs written in complete C, Fortran, ML, and Scheme. Other conventional analyses (dependence analysis[2], alias analysis[4]) can also be implemented by Z2.

In this article, we report our on-going work to adapt Z2 to generating SUIF program analyzers. Together with this adaptation, a graphic user interface is also being developed, which will enable the users to see the input program and its analysis result simply by clicking the mouse. The end-result will be an integrated environment for developing global, interprocedural, semantic-based analyzers for SUIF programs.

2 Generating Collecting Analyzers

Z2 generates a program analyzer which computes, for each program point, information which describes the possible program states at that points during program execution. We call this *collecting analysis*.

An interpreter is a function to represent the evaluation rule for each program construct. Each evaluation rule is a state transformer: a function from pre-state to post-state. An abstract interpreter $\overline{I}$ for target language is defined as followed type:

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

where Σ is a set of program points, and $\overline{X}$ and $\overline{Y}$ are lattices of pre-state and post-state. The abstract interpreter is defined as follows:

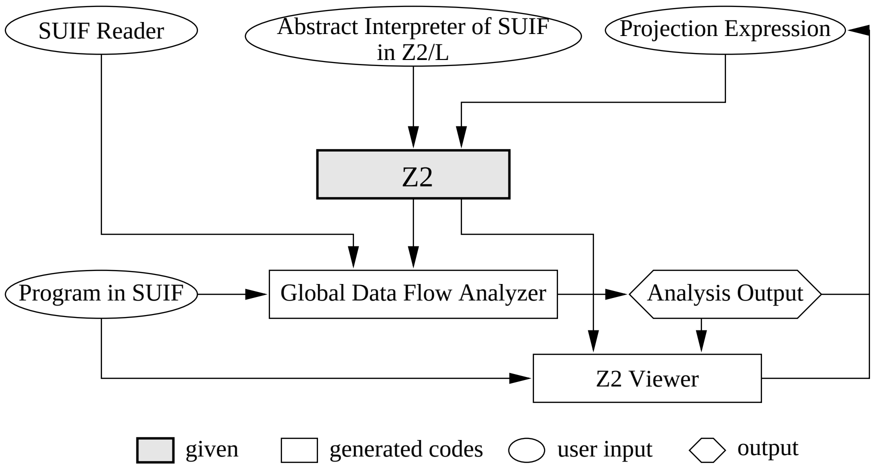

Figure 1: The System Z2

$$\begin{aligned}\overline{I} = \lambda\sigma.\lambda\overline{x}.\ &\text{case } (\sigma) \text{ of}\\ &\quad \sigma \text{ an if-statement: } \overline{I}_1(\sigma, \overline{x})\\ &\quad \sigma \text{ a procedure call: } \overline{I}_2(\sigma, \overline{x})\\ &\quad \ldots.\\ &\quad \sigma \text{ a constant: } \overline{I}_n(\sigma, \overline{x})\end{aligned}$$

where $\overline{I}_i$'s are evaluation rules for each program construct and usually involve recursive calls of $\overline{I}$.

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

$$Tabulate(F_{\overline{I}}, \Sigma_P, \overline{x}_0)$$

where $F_{\overline{I}}$ is the associated functional of the recursive definition of $\overline{I}$, that is, $F_{\overline{I}} = \lambda\overline{I}.\lambda\sigma.\ldots\overline{I}\ldots$, Σ_P is the set of program points of a program P, and $\overline{x}_0$ is the start pre-state.

The analysis result is two tables $T_{\overline{X}}$ and $T_{\overline{Y}}$. These tables represent, for each program point σ, the pre-state $T_{\overline{X}}(\sigma) \in \overline{X}$ and the post-state $T_{\overline{Y}}(\sigma) \in \overline{Y}$, which describe the states that occur before and after that point during execution. The *Tabulate* function computes the fixpoint of abstract interpreter definition by iteration for program points until $T_{\overline{X}}$ and $T_{\overline{Y}}$ are not changed.

3 Integrating Z2 with SUIF: An Example

In the followings, we briefly present, as an example, how we specify an interprocedural constant propagation analysis for SUIF programs: defining SUIF parser interface for Z2, specifying the analysis as an abstract interpreter of SUIF.

3.1 Interface between Z2 and SUIF

Since System Z2 does not generate the syntax tree, nor does it provide any operator to access its nodes, the user must write C or C++ procedures for these operations. We wrote C++ procedures to access SUIF nodes.

```
POST = V * S                  ; post-state
PRE = S                       ; pre-state

V = L * Z                     ; value
S = Id -> V                   ; store

L = power setId               ; location
Z = power setZ                ; integer value

Id = lift setId               ; identifier

setId = [0 : NumOfIds()]      ; set of identifier
setZ = [-99999 : 99999]       ; set of integer
```

Figure 2: Abstract domain specification

First, we connect a reserved data type `syntree` in Z2, which represents the nodes of abstract syntax tree node, to SUIF syntax tree. We define `syntree` as pointer to `suif_object` which is the most general class in SUIF. Some kinds of nodes in SUIF tree structure such as `tree_node_list` or `operand` are not derived from `suif_object`. So, we define new annotations derived from `suif_object`, which point to heterogeneous nodes, and inserted them to their parent nodes.

Then, we define some functions to access SUIF nodes. We give a unique number to each abstract syntax tree node and each symbol for interface between SUIF nodes and their indices inside the abstract interpreter. Functions `SyntreeToId()` and `IdToSyntree()` returns the corresponding number or syntax tree node from syntax tree node or number. Function `Xsym()` returns a corresponding number from symbol node. We also need to implement functions to recognize the kind of SUIF nodes and to get their child nodes. Functions such as `isTreeBlock(), isTreeFor(), isAdd()`, etc distinguish between the kinds of nodes, and functions such as `IfThen()`, `LoopBody()`, etc retrieve the child nodes.

These definitions and declarations are defined in files `user.c` and `user.h`, which are compiled with the other files generated by Z2.

3.2 Abstract interpreter specification

We use a specification language Z2/L to specify the abstract interpreter. It consists of three parts: one for domain specification, one for semantic function specification, and one for projection to simplify designed domains.

Our abstract interpreter of SUIF takes an abstract store(`S`) and returns a tuple of abstract value(`V`) and the next store(`S`). Value has storage locations(`L`) or integer values(`Z`), which are, respectively, defined as powersets of set of identifiers(`setId`) and set of integer(`setZ`). An element of `V` is a collection of locations or integers that each program point can has during execution. The store(`S`) is a mapping from identifiers(`Id`) to value(`V`). It represents values that every identifier has. Figure 2 is a domain specification for SUIF in Z2/L. `*`, `->`, `power`, and `lift` represent product lattice, function lattice, powerset lattice, and flat lattice, respectively. $[z_1 : z_2]$ represents an integer range set from z_1 to z_2.

Abstract interpreter function `Eval`, which operates on the designed domain, returns post state(`V * S`) from pre state(`S`) for each syntax tree node. `Eval` is recursively defined for each construct of SUIF. Figure 3 shows a part of the `Eval` function and one of its auxiliary function.

Projection expressions simplify the designed abstract interpreter without modifying other parts of specification. Z2 uses the height of element in the lattice as projection condition. Since elements which have more than two integers have heights of over two, projection statement `Z = height() > 1` projects the elements that is not a constant. Another projection `L = height() > 5` is used in order to reduce analysis cost. It makes analyzer less accurate in less cost.

```
;;;  interpretation function for for-loop body
fun foreval(x:syntree, s:S, id:V, curindex:V, upper:V, step:V):POST =
        let
          val test = SleV(curindex, upper)
        in
          if 1:setZ in test.Z then
            let
              val s1 = join fn(l:setId):S => (s[curindex/l:Id]) for l in id.L
              val post1 = Eval(x, s1)
            in
              post1 join foreval(x, post1.S, id,
                                  AddV(curindex, step), upper, step)
            end
          else
            <bottom:V, s>:POST
        end

;;; main abstract interpreter function for SUIF
semantics Eval(x:syntree, s:S, e:E):POST =
  case
    isTreeInstr(x) =>
        ....
  | isTreeloop(x) =>
        ....
  | isTreeIf(x) =>
        ....
  | isTreeFor(x) =>
        let
          val idx = Xsym(ForIndex(x)):setId:Id
          val lb = Eval(ForLb(x), s, e)
          val ub = Eval(ForUb(x), s, e)
          val step = Eval(ForStep(x), s, e)
        in
          foreval(ForBody(x), s, e, idx, lb, ub, step)
        end
  | isNop(x) =>
        ....
    ....
```

Figure 3: Abstract interpreter specification for SUIF

3.3 Analysis result

Figure 4 is an example of collecting analysis of SUIF. (a) is a SUIF program including IF

`1: ldc i = 2`		
`2: ldc j = 3`	i = { 2 } j = { 3 }	i = { 2 } j = { 3 }
`3: IF (Jumpto=L:L1)`		
`IF HEADER`		
`4: bfalse e1, L:L1`		
`5: e1: sne n, e2`		
`6: e2: ldc 0`		
`IF THEN`		
`7: add i = i, j`	i = { 5 } j = { 3 }	i = { 5 } j = { 3 }
`IF ELSE`		
`8: ldc i = 0`	i = { 0 } j = { 3 }	i = { 0 } j = { 3 }
`IF END`	i = { 0, 5 } j = { 3 }	i = $\top$ j = { 3 }

(a) SUIF program (b) Collecting analysis result (c) Simplified result by projection

Figure 4: Analysis results

construct. (b) shows the values which identifiers `i` and `j` can be at each point during execution. (c) is the result after projection by which all elements that have more than two values are projected to top. We can convert the expressions which have a unique value in their post-state to constants.

Graphic user interface(ZV) makes it easy to see the analysis result at each program point. We can see the state of a program point by clicking the mouse on the corresponding point in source code showed in window. Figure 5 shows a sequence(represented as arrows) of windows that are popped up to examine an analysis result at a program[1] point. Note that graphic user interface modules are also generated by Z2, because the interface must vary depending in the lattice structures of the specified analysis.

4 On-going Works

The Z2 environment for SUIF programs clearly has several benefits. By integrating Z2 with the SUIF system, the data-flow analyzer designers can see the "real" effectiveness of their analyzers, because the analysis result can be easily fed to the later compilation phases to be reflected in the final object code. Implementing multiple versions of a data flow analysis for different target languages are not necessary. Z2 bridges the gap between the formal semantics-based analyses and popular, real-world programs in complete C, C++, Fortran, Java, and etc. that SUIF supports.

References

[1] Liling Chen, Luddy Harrison, and Kwangkeun Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 3(1):31–68, 1995.

[2] Jyh-Herng Chow. *Compile-Time Analysis of Explicitly Parallel Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[1]The program is written in a language that we used in our senior compiler course

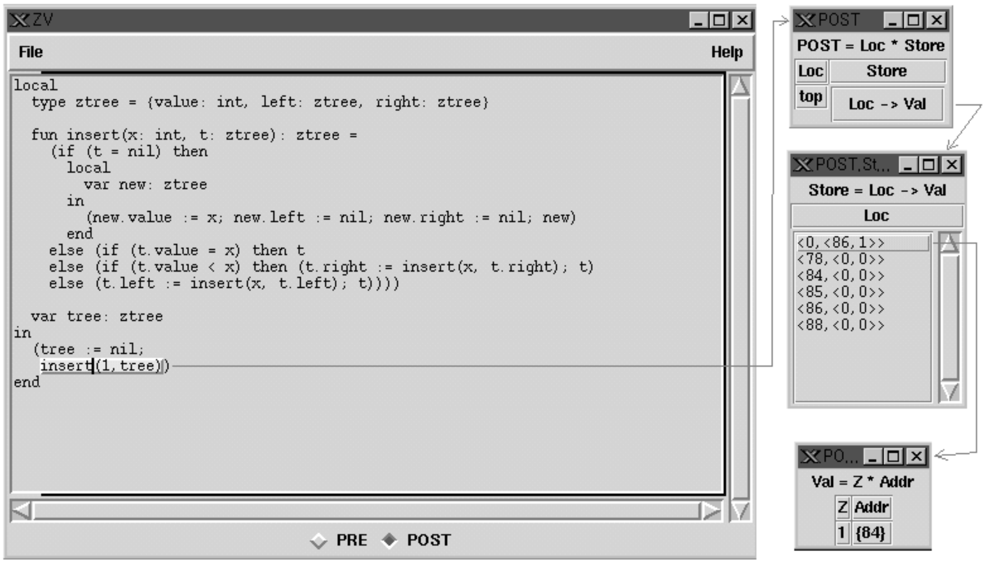

Figure 5: Graphic user interface for Z2

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceeding of Fourth Annual ACM Symposium of Principle of Programming Languages*, pages 238–252, 1977.

[4] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM Conference on Programming Languages Development and Implementation*, 1994.

[5] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analysis. In *Proceeding of 20th Annual ACM Symposium of Principle of Programming Languages*, pages 246–259, 1993.

[6] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in sml programs. In *Proceedings of the 4th International Static Analysis Symposium*, Lecture Notes in Computer Science, 1997.