Automatic Generation and Management of Program Analyses

Kwangkeun Yi
Department of Computer Science
Center for Supercomputing Research and Development[1]

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

[1]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.

ABSTRACT

Designing a program analysis (for use in optimizing compilers) is a time-consuming, complicated process because it involves many parameters. In particular, one must carefully measure the accuracy of the analysis against its cost. To reach an acceptable cost-accuracy balance the designer must repeatedly implement the analysis, measure its behavior, and alter its design.

In this thesis we present a software system called system Z1, by which one can quickly specify a program analysis, obtain automatically from this specification an executable version, measure its performance, and alter the design (adjust the cost-accuracy balance), and repeat the cycle until the desired behavior is seen. Specification language Z1 is designed in which the user can conveniently express various program analyses for a wide range of target languages. System Z1 is based on the abstract interpretation framework. The input to system Z1 is a high-level specification of an abstract interpreter. The output is a C code for the specified interprocedural program analyzer.

What makes system Z1 especially useful is called a projection expression. The user writes down, at the outset, a straightforward specification of an analysis, essentially ignoring its cost (memory and cpu time) required to compute it. Afterwards, projection expressions are used to control the cost-accuracy tradeoff of the analysis. The user adds projection expressions for selected domains. System Z1 automatically generates a new analyzer which will conduct the program analysis over the simplified domain structure. Projection expressions permit accuracy to be traded against cost, without requiring that the specification of the analysis be altered.

Using system Z1 we have designed constant propagation, alias analysis, def-use/use-def chain analysis, and memory object lifetime analysis. The target languages are C, FORTRAN, and SCHEME, without restriction upon the programs treated. We presents a series of experiments results that demonstrate the uses of projection expressions for these analyses problems.

To my parents

ACKNOWLEDGMENT

First, I thank my advisor Williams Ludwell Harrison III. I was very, very lucky to be trained under such a great supervision. His enthusiasm and brilliant ideas stimulated me in every step of my research. His understanding and patience were always there whenever I need them.

I thank the members of my thesis committee: Samuel Kamin, David Kuck, Dennis Mickunas, and David Padua. I thank Samuel Kamin for several suggestions about my research. And, without his assistance during the POPL '93 symposium I would not be able to have a chance to work with David MacQueen's group at Bell Labs after graduation. In this regard, I thank Uday Reddy too. I thank David Padua. I decided to specialize in compiler after I took his introductory course in parallel programming. When I asked for his suggestion regarding my graduate research he helped me be the first student of Luddy.

It was a bless that I could work in such wonderful an environment like the Center for Supercomputing Research and Development (CSRD). I thank all its administrative staff. In particular, I thank Linda Ashby for those insurmountable copy jobs wonderfully done.

I also wish to thank my colleague graduate students: Jyh-Herng Chow, Li-Ling Chen, Mohammad Heghighat, Sharad Mehrotra, Sirpa Saarinen, and Mei-Chen Tsai. Jyh-Herng kindly spent many hours enlightening me both in research and daily life.

I thank my wife. I cannot image how my life, without her, might have been drifted during my college and graduate years. It was a pleasant coincidence that my research started to ripen when we were expecting our son. I also thank Korean community of this town for the many hours they shared with us.

Finally, of course, I want to thank my parents and family members. I would just like to keep in my heart a deep gratitude to them; I am too grateful to say anything else.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem and Goal

Semantic analysis, and especially interprocedural analysis, is an important component of compilers for modern computer systems. Yet, building a program analysis (for use in optimizing compilers) is a time consuming, complicated process because it involves many parameters. In particular, one must carefully measure the accuracy of the analysis against its computation cost. When an analysis performance is not acceptable (because it is too costly or too inaccurate), a new analysis, which will differ in its cost-accuracy balance from the previous one, must be designed, re-implemented, and tested again. Until the desired behavior is observed one must repeat this cycle.

Our goal is to design a software system by which one can quickly achieve an accurate yet affordable program analysis. The black box in Figure 1.1 represents the tool. Its user writes an analysis specification and sets the dial for a desired cost-accuracy tradeoff. The tool then creates an executable analyzer which has the specified performance balance.

This problem involves several aspects:

- The software system needs a high-level specification language in which one can conveniently express a various kinds of analyses for a wide-range of target languages.
- It should provide a simple mechanism by which the user can easily adjust the accuracy-cost tradeoff of his analysis without much efforts (without discarding the entire analysis and beginning anew).
- It should be able to produce from the specification an acceptably efficient executable, because while it is not necessary for the generated analysis to run as fast as hand-crafted one, it must permit reasonable experiments to be run, so that various degrees of accuracy may be compared.

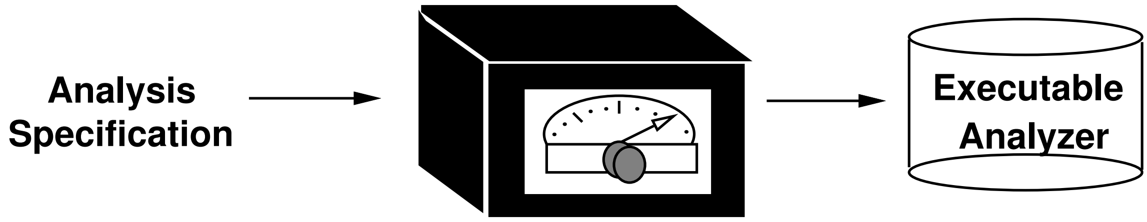


Figure 1.1: The Goal

1.2 Thesis

We developed a software system called *system Z1* that fits the descriptions mentioned above.

System Z1 automates the implementation of a program analysis *and* the management of the cost-accuracy balance of the analysis. System Z1 is a language processor that translates the input specification into an executable analyzer. The input specification consists of two parts: specifications for the analysis and for its cost-accuracy tradeoff. From this input, the system generates a program analyzer in C. The system arranges that the generated analyzer should conduct the analysis with the specified cost-accuracy tradeoff.

System Z1 is based on *abstract interpretation* [CC77, CC79, CC92a]. The input for the generation part is an abstract interpreter of the target language. The input for the management part is a set of expressions (called projection expressions) that controls the degree of abstraction of the interpreter. Because the abstract interpretation framework views program analysis as a compile-time simulation of actual execution, some modern languages (like C and SCHEME), which are considered difficult to analyze by the traditional data flow analysis framework, can be analyzed without much difficulty. Moreover, because it forces analysis designers to work rigorously with the semantics of the target language, it keeps complicated analysis from violating the original semantics of the input programs.

The input specification language enables us to specify a wide range of program analyses for various target languages. We were able to develop both the conventional program analyses (like alias analysis, constant propagation, use-def/def-use chain analysis) and the non-conventional, aggressive ones (like interprocedural side-effects analysis, memory object lifetime analysis, and cyclic structure detection) with C, FORTRAN, and SCHEME as their target languages.

System Z1 is a serious tool for rapid prototyping of non-trivial, detailed semantic analyses of programs. It addresses and treats the most difficult programming language constructs in use. We cannot, however, claim that the generated analyzer is as good as hand-crafted one. A good programmer can write a better (in performance) implementation for the same analysis specification than system Z1 does.

1.3 Interprocedural Analysis As a Separate Entity

Interprocedural analysis is to analyze the programs that contain procedure calls. This introduces new problems that are not encountered when the analysis is restricted to a single procedure body. Under abstract interpretation, however, interprocedural analysis is *not* a problem that requires a separate analysis technique from intra-procedural one. It seems like all that matters is which analysis framework we will use. Before we elaborate this claim, let us review a background of how interprocedural analysis came its way.

Traditional data flow analysis methods (to cite some, [Hec77, Ros80, Tar76, KU77, KU76, GW76, JM82, Ken76, Ros77, CCF91, MR90, CR88, RP86]) did not pay much attention to the semantics of the target programs. This is perhaps because their target language was usually FORTRAN, in which a program's meaning was largely manifest in its text. Thus, it was straightforward to set up the data flow transition function at each program node, because all necessary information would be explicit in the program text.

However, one expression in FORTRAN whose effect is not exposed in its text is the procedure call. In order to derive the information about its effect one must examine the body of the called procedure. This introduces some complications. Because of the reference parameters one must consider the alias relation between variable names across procedure boundaries. Summary information concerning the procedure body must also be able to handle direct or indirect recursive calls. Studies that addressed these issues are traditionally called "interprocedural data flow analysis"[Cal88, Mye81, Eor91, Bar77, Wei80, CCKT86, BC86].

In more modern programming languages the procedure call is not the only expression whose effect is not evident in its text. For example, in C, unrestrained pointer manipulations, dynamically generated memory, and procedure pointers as parameters can easily compose expressions whose effect cannot be derived by a simple inspection of their text. It seems that the dichotomy of interprocedural and intra-procedural analysis is an unnatural one, because we may need many other specialized analysis methods for each of such language constructs: pointer analysis for C, closure (first-class function) analysis for SCHEME, method binding analysis for C++, and so on.

There is, however, one thing common to this pile of analysis techniques. Every such technique depends upon the semantics of the involved constructs. They must take the semantics into account in order to understand the effects of the problematic constructs. In other words, each of them must, in some way, simulate the executions of the construct to analyze expressions written using them. This is exactly the approach endorsed by the *abstract interpretation* framework. In this framework, designing an analysis is equivalent to designing an approximated interpretation rule of the target language. Compile-time interpretation rules for the language constructs are all we need. Whether an expression's effect is explicit in its text or not is then not a problem. Pointer expressions, upward function arguments, etc. are handled by the interpretation rules for those constructs. By the same token, interprocedural analysis is naturally handled by the abstract evaluation rules for procedure call expression. Interprocedural analysis in the abstract interpretation does not require a separate analysis technique.

1.4 Structure of the Dissertation

This dissertation is organized as follows:

- The rest of this chapter overviews system Z1. We discuss its input/output relation and the feedback from the user to manage an existing analysis' cost versus accuracy tradeoff.
- Chapter 2 discusses how an abstract interpreter definition is understood by system Z1 as a specification for a program analysis. We define *collecting analysis*, which we mean by "program analysis". We discusses one framework for defining a correct abstract interpreter. We introduce *projections* and give an intuition about how they change accuracy and cost behavior of an existing analysis. We also summarize the reasons why we choose abstract interpretation as a framework behind this tool.
- Chapter 3 defines the space $\mathcal{L}$ of abstract domains that is supported by Z1. We discuss several important properties preserved by the domains of $\mathcal{L}$. These properties are essential for *projections* and economical implementations of the program analysis.
- Chapter 4 defines the abstract interpreter specification language Z1. It include the constructs for defining abstract domains of $\mathcal{L}$, those for defining projections, and those for defining the abstract interpreter function. We present a toy example of an abstract interpreter specification for expository purposes.
- Chapter 5 presents a series of analyses that are designed using Z1. The target languages of the analyses are C, FORTRAN, and SCHEME. We experiment with projections to demonstrate how can they change an analysis in accuracy and cost.
- Chapter 6 presents our conclusions. We review the related work of other researchers and discuss the current limitations of Z1.

1.5 System Overview

The overall configuration of system Z1 is shown in Figure 1.2. A more detailed block diagram is shown in Figure 1.3.

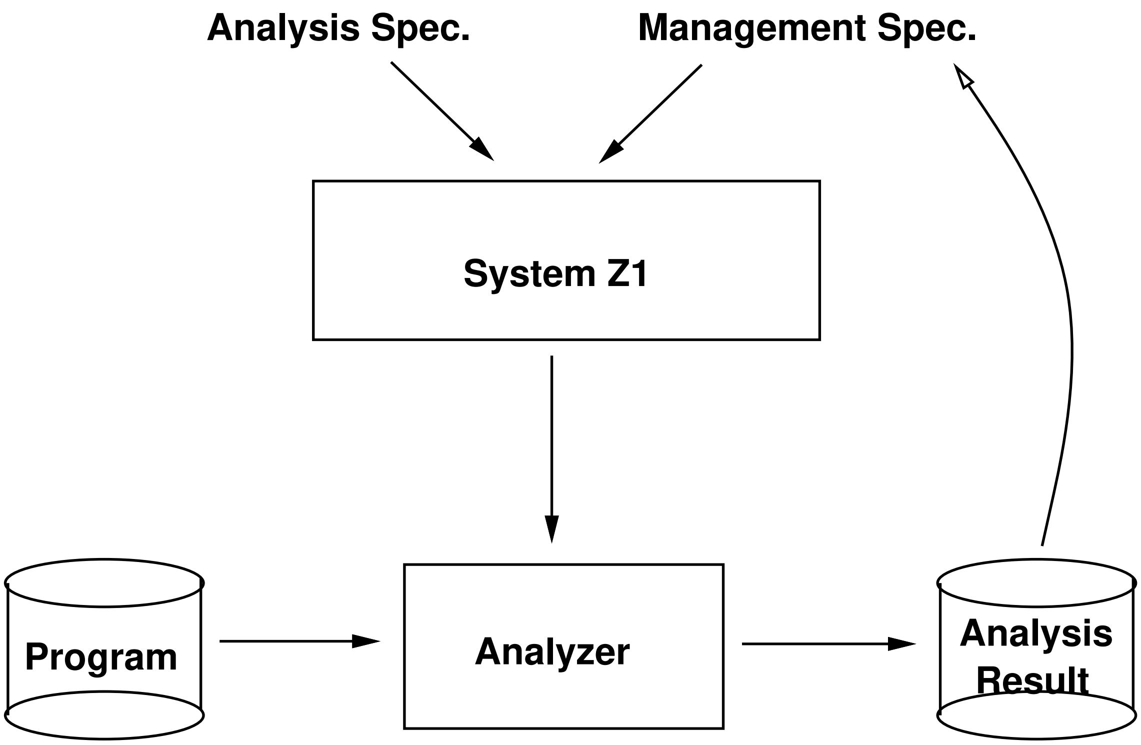


Figure 1.2: The System Z1

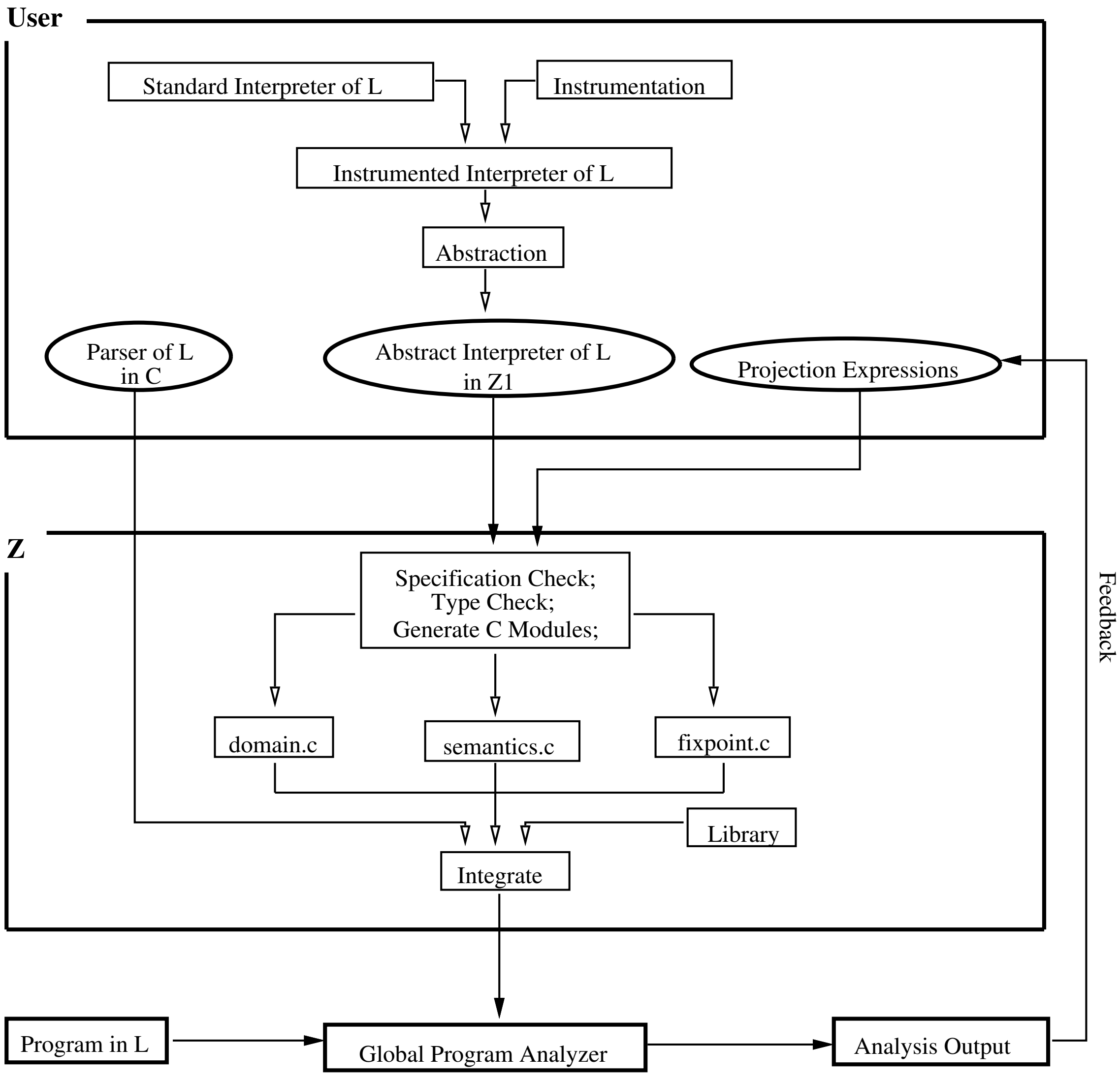


Figure 1.3: The System Z1 (a detailed diagram)

1.5.1 Analysis Generation

The input to system Z1 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 guaranteed always to terminate. This abstract interpreter specification is the starting point for (input to) system Z1.

From an abstract interpreter specification, system Z1 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).

1.5.2 Analysis Management

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 Z1 provides a simpler facility: *projection expressions*. By this expression set the user can specify the cost and accuracy tradeoff. The more projection the less accurate (and less costly) a generated analysis will be. The user writes projection expressions for an existing abstract interpreter definition. System Z1 automatically generates an analysis which will process the input programs with the cost and accuracy tradeoff specified by the projection expressions.

1.5.3 What Z1 Can Do

- From a high-level specification, Z1 creates a C code for an interprocedural program analysis.
- From a high-level specification, Z1 tunes the analysis in cost and accuracy.

- Z1 is based on the abstract interpretation framework. Thus, its virtues come from the power of the abstract interpretation framework [CC77, CC79, CC92a]. Z1 allows us to specify a detailed analysis for difficult and dynamic language features (e.g., aliases, unrestrained pointer manipulation, dynamic memory allocation, first-class functions). This is because a program analysis in the abstract interpretation framework is viewed as an approximated simulation of input programs.
- Z1 provides the run-time support (e.g. garbage collection), and generates utility procedures (e.g. data type pretty printers, dump/restore routines for the analysis result, debugging aids) that are needed to examine the analysis result and to debug the input specification.

Z1 has been used for constant propagation, alias analysis, def-use/use-def chain analysis, procedural side-effect analysis, and memory object lifetime analysis. The target languages were C, FORTRAN, and SCHEME without restrictions upon the programs treated. We will present these analyses in Chapter 5.

1.5.4 What Z1 Cannot Do

- Z1 cannot *design* an analysis. The analysis design must be done by the user. The user specifies the analysis in the specification language provided by Z1.
- Z1 does not provide the parser for a target language.
- Z1 succumbs to the limitations of the specification language. We have designed the specification language to be general enough to specify a large class of 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 variety of analyses are tried; we plan to expand the language if necessary.

1.5.5 Snapshots

We present three snapshots of using Z1 in Figure 1.4, Figure 1.5, and Figure 1.6.

Figure 1.4 shows the compilation of an input specification into three C modules by the system. The input specification file is `cp-alias1.z`, which describes constant propagation and alias analysis. These two analyses are embodied in single specification. The input has 522 lines. The generated C code has 5370 lines in total.

Figure 1.5 shows the compilation of the generated C code. The compiled code is linked with the target language parser and the system library. After this process, the executable (named `zil`) size becomes 278K bytes.

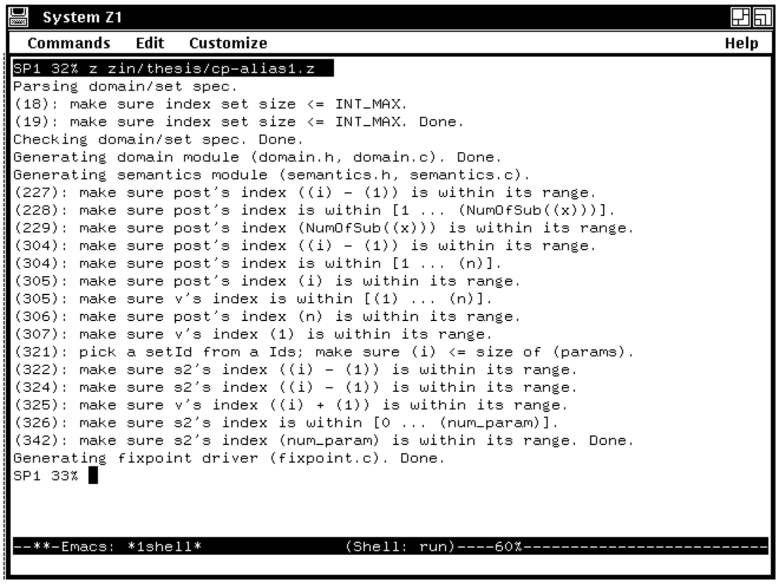
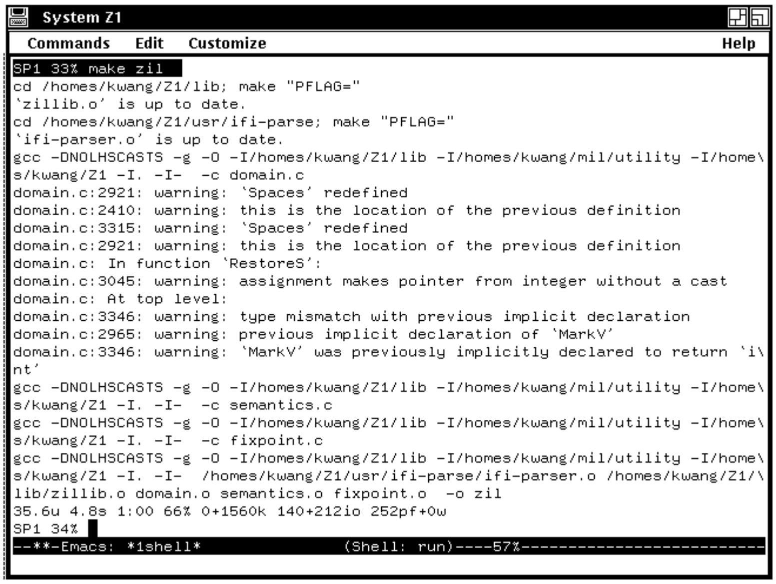


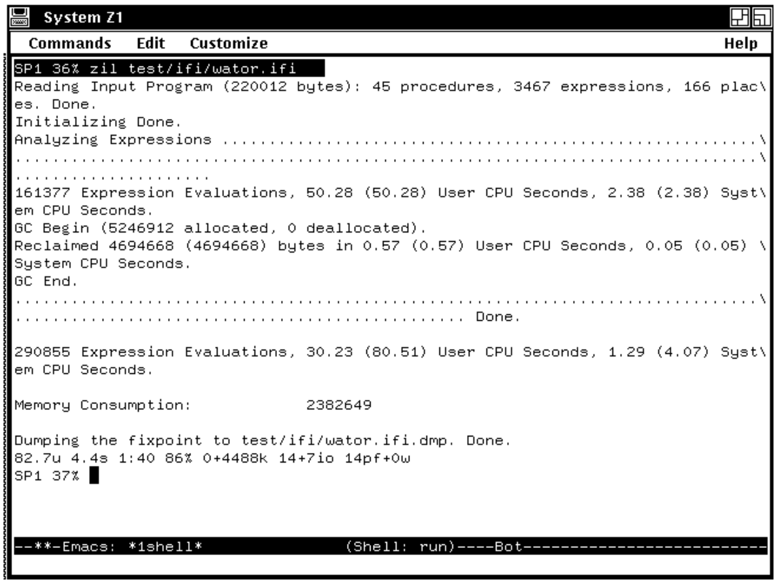
Figure 1.4: Z1 Processes the Input

Figure 1.6 shows the analysis computation of the executable analyzer for an input program. The input program `wator.ifi` is an ecological simulation program written in C (448 lines). This C program is translated into an intermediate language before the analysis; this is why the file extension is `.ifi`, not `.c`. This snapshot shows that during the analysis garbage collection is invoked twice.



```
System Z1
Commands   Edit   Customize                                              Help
SP1 33% make zil
cd /homes/kwang/Z1/lib; make "PFLAG="
`zillib.o' is up to date.
cd /homes/kwang/Z1/usr/ifi-parse; make "PFLAG="
`ifi-parser.o' is up to date.
gcc -DNOLHSCASTS -g -O -I/homes/kwang/Z1/lib -I/homes/kwang/mil/utility -I/home\
s/kwang/Z1 -I. -I-  -c domain.c
domain.c:2921: warning: `Spaces' redefined
domain.c:2410: warning: this is the location of the previous definition
domain.c:3315: warning: `Spaces' redefined
domain.c:2921: warning: this is the location of the previous definition
domain.c: In function `RestoreS':
domain.c:3045: warning: assignment makes pointer from integer without a cast
domain.c: At top level:
domain.c:3346: warning: type mismatch with previous implicit declaration
domain.c:2965: warning: previous implicit declaration of `MarkV'
domain.c:3346: warning: `MarkV' was previously implicitly declared to return `i\
nt'
gcc -DNOLHSCASTS -g -O -I/homes/kwang/Z1/lib -I/homes/kwang/mil/utility -I/home\
s/kwang/Z1 -I. -I-  -c semantics.c
gcc -DNOLHSCASTS -g -O -I/homes/kwang/Z1/lib -I/homes/kwang/mil/utility -I/home\
s/kwang/Z1 -I. -I-  -c fixpoint.c
gcc -DNOLHSCASTS -g -O -I/homes/kwang/Z1/lib -I/homes/kwang/mil/utility -I/home\
s/kwang/Z1 -I. -I-  /homes/kwang/Z1/usr/ifi-parse/ifi-parser.o /homes/kwang/Z1/\
lib/zillib.o domain.o semantics.o fixpoint.o  -o zil
35.6u 4.8s 1:00 66% 0+1560k 140+212io 252pf+0w
SP1 34%
--**-Emacs: *1shell*            (Shell: run)----57%-------------------------
```

Figure 1.5: Compiling the Generated Analyzer and Linking It with the Library



```
System Z1
Commands   Edit   Customize                                              Help
SP1 36% zil test/ifi/wator.ifi
Reading Input Program (220012 bytes): 45 procedures, 3467 expressions, 166 plac\
es. Done.
Initializing Done.
Analyzing Expressions .......................................................\
.............................................................................\
.....................
161377 Expression Evaluations, 50.28 (50.28) User CPU Seconds, 2.38 (2.38) Syst\
em CPU Seconds.
GC Begin (5246912 allocated, 0 deallocated).
Reclaimed 4694668 (4694668) bytes in 0.57 (0.57) User CPU Seconds, 0.05 (0.05) \
System CPU Seconds.
GC End.
.............................................................................\
............................................... Done.

290855 Expression Evaluations, 30.23 (80.51) User CPU Seconds, 1.29 (4.07) Syst\
em CPU Seconds.

Memory Consumption:            2382649

Dumping the fixpoint to test/ifi/wator.ifi.dmp. Done.
82.7u 4.4s 1:40 86% 0+4488k 14+7io 14pf+0w
SP1 37%

--**-Emacs: *1shell*            (Shell: run)----Bot-------------------------
```

Figure 1.6: Generated Analyzer Analyzes a Source Program

Chapter 2

Program Analysis From an Abstract Interpreter Definition

In this chapter, we will present how system Z1 compiles an input specification into a corresponding program analysis. The generated program analyzer will compute, for each program point, a pre-state and a post-state that can occur before and after the program point during execution. We will discuss the problem of managing the cost versus accuracy tradeoff of the program analysis and system Z1's solution to this problem. We will also discuss the reasons behind our choice of an abstract interpreter as an input to the system.

2.1 Input to Z1: An Abstract Interpreter

Z1 generates a global program analysis from a high level specification of an analysis. The specification is an *abstract interpreter* definition. This abstract interpreter specification is the starting point for (input to) system Z1.

The abstract interpretation framework [CC92a, CC79, CC77, AH87] is a method for designing approximate interpreters which can provide sound answers to questions about the run-time behavior of programs. An abstract interpreter determines how to simulate the programs correctly at compile-time (that is, without overlooking any of their run-time behavior). This method is a unified approach to program analysis because every analysis problem is concerned with the run-time behavior of programs, which can be simulated by a properly designed abstract interpreter. Consider dependence analysis. This requires that we determine, at compile-time, which memory portion is defined and later read by which program expressions. This information can be collected if we interpret (at compile-time) the program, and record the memory locations that each expression accesses each time a value is read or written. As another example, consider constant propagation analysis. Expressions that remain constant during execution will be discovered if we interpret (again at compile-time) the programs recording the values that each expression is evaluated into.

There are two special things about an abstract interpreter. It must be approximate, because it is to be used at compile-time. Though it is approximate, it must be safe. The approximate interpretation errs conservatively (that is, it does not omit any run-time behavior). Finally, since it is compile-time interpretation, it must terminate even when the program might not. The user is responsible for defining a *correct* abstract interpreter. System Z1 does not check the correctness of an abstract interpreter definition with respect to a standard or instrumented semantics. It only checks its type consistency and, if it is type safe, generates a program analyzer. In Section 2.1.2 we will review a framework for defining a correct abstract interpreter. Before that, we shall first discuss why we chose abstract interpreters as inputs to system Z1.

2.1.1 Why Abstract Interpretation?

The virtues of system Z1 come from the power of abstract interpretation.

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

  Conventional data flow analysis methods (as summarized in [RP86, Hec77]) have several limitations.

  First, they assume the control flow graphs of programs prior to the analysis. For languages like FORTRAN this is not a problem because the control flow graph of a program can be (largely) determined from its text. However, this does not hold for languages like C or SCHEME. In these languages, procedures can be passed as parameters (or returned as results of other procedures) and thus they are not always invoked by their defined names. Suppose a program contains the assignment

  $$\texttt{f(g)} = \cdots \texttt{g(x)} \cdots$$

  From the text, we cannot know to which procedures g may be bound. If we are to have the control flow graph before the analysis, we cannot help assuming that g is bound to every procedure, which will cause expensive and inaccurate analysis. In order to avoid this overly conservative approach, we must follow the program execution and simulate the parameter binding mechanism. But this requires as complicated a computation as another program analysis! This problem is well-known and addressed in several studies [Shi91, Shi88, HK92]. Shivers [Shi91, Shi88] used continuation passing style representations of SCHEME programs to simplify his control flow analysis. Hall and Kennedy [HK92] devised a compile-time method of simulating the parameter binding mechanism to compute the call graph of C or SCHEME programs. In the abstract interpretation, however, determining the control flow graph is not a prerequisite for an analysis. Because abstract interpretation is a compile-time "simulation" of programs, it needs only a compile-time interpretation rule for each language construct and the source program text. The program's control flow graph is naturally followed during an interpretation of the source program.

Second, only with difficulty can the conventional data flow analysis methods be adapted to constructs that require detailed semantic consideration. For languages like C, not all accessible memory locations are named explicitly when they are accessed. In

```
*(x+C)
```

the variable `x` does not (syntactically) specify which memory location it points to. It is because of the unrestrained pointer manipulations, aliases, and dynamic memory allocation supported by C. Unless we interpret those program constructs, we must assume that `x` points to every memory location. Under abstract interpretation, all we need is an abstract evaluation rule for memory access. Compile-time simulation of those constructs will then compute which memory location they access.

Another important virtue of an abstract interpretation is that it is derived from 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. This is important because the current trend requires an increasingly aggressive compile-time optimizations for modern computer systems.

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

  Under abstract interpretation, interprocedural analysis is no more difficult than intraprocedural analysis. An approximate, compile-time interpretation rule for procedure calls 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 [Cal88, Mye81, Wei80, Bar77] in two respects. 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, the 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 [Cal88, Mye81] 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 [Sto77], 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.

However, 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 a correct, detailed semantic analysis (whether it be interprocedural or intra-procedural) for a broad class of target languages.

As an input to an automatic tool for generating program analysis, the abstract interpretation method is hard to defeat:

- It can support virtually any analysis for any target programming languages. All we need is an appropriately defined abstract interpreter for the language.
- Analysis cost vs accuracy tradeoff can be controlled by the degree of abstraction.
- An appropriate formalism for expressing the abstract interpreter rules is the input language to the automatic tool.

2.1.2 Abstract Interpreters in Z1

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. Below we will discuss what is required to construct a correct abstract interpretation under Z1.

Defining Correct Abstract Domains

Abstract domains are the abstract value space over which the abstract evaluation rules operate. Defining a correct abstract domain involves two things:

- Defining each abstract element's meaning
- Constructing a relation between abstract elements

A clear idea about what each abstract element means is essential because, without it, the abstract interpreter designer cannot define a meaningful evaluation rule nor can he derive useful information from the evaluation result. Constructing a relation between the abstract elements is also necessary because, during program analysis, we must choose a safe element which is properly "related" to the current elements in hand.

Each abstract element denotes a (possibly infinite) set of concrete values. Concrete values are the entities that occur during real executions of programs: integers, reals, strings, procedures, variables, records, pointers, stores, environments, etc. A precise mathematical model for these entities is provided by *denotational semantics* [Sto77, Sco72, Sco70]. This mathematical semantics makes "visible" [Sco70] what we have in mind about what these entities are. The precise semantics of concrete values determines the precise meanings of the corresponding abstract values.

In denotational semantics [Sto77], a mathematical structure called a *domain* is used to give mathematical meaning to concrete values. A *domain* is a *pointed, complete, partially ordered set.* Pointedness requires an element (bottom) which is lower than all other elements. Completeness requires that the least upper bound of each chain exists in the set. In this model, a concrete value denotes the limit of a chain in a domain. An instrumented interpreter, from which we derive its abstract counterpart, is defined in the denotational semantics formalism.

Since an abstract value denotes a set of concrete values, we need some entity that accommodates a collection of concrete meanings. Such entity is one called *ideal.* Here, an abstract value denotes an ideal of the corresponding domain in the concrete, instrumented interpreter. An ideal J of a domain D is a subset of D 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). Note that an ideal models more than one concrete value because several chains can be included in single ideal.

The relation between an abstract value $\overline{x}$ and its meaning (an ideal) $\gamma(\overline{x})$ is described by the adjoined pair:

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

Note that, in the computer during analysis, each element of an abstract domain may be represented as a single token. The mathematical meaning of each token is not realized in this representation; it exists in the analysis designer's mind.

Each abstract value space is structured into a finite-height lattice, which is ordered by set inclusion of the ideals that constitute its abstract elements. The finite-height requirement guarantees the termination of the induced analysis. The lattice structure guarantees that for every pair of abstract values their greatest lower bound and least upper bound exist.

Defining Correct Abstract Evaluation Rules

In denotational semantics, an interpretation rule for a language construct is modeled as a *continuous function* $f: A \rightarrow B$ over the domains A and B. The corresponding abstract operation $\overline{f}$ must be a upper approximation of f:

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

The result $\overline{f}(\overline{x})$ over $\overline{x}$ must include the concrete result $f(x)$ for every x meant by $\overline{x}$. (Recall that $\overline{x}$ and $\overline{f}(\overline{x})$ represent ideals of A and B, respectively.) This requirement guarantees the correctness of an abstract interpreter with respect to its concrete correspondent. Also, every abstract evaluation rule $\overline{f}$ must be sensitive to the relation of its arguments in a special way. It must be monotonic:

$$\overline{x} \sqsubseteq \overline{y} \Longrightarrow \overline{f}(\overline{x}) \sqsubseteq \overline{f}(\overline{y})$$

The more concrete values its input represents, not less concrete values its output represents. This, together with the finite-height of lattices, guarantees the termination of the induced program analysis.

System Z1 supports this formalism for defining abstract interpreters.

System Z1 provides a set of constructs to define finite-height lattices for abstract domains and a set of constructs to define functions for abstract evaluation rules. **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 Z1. The user is recommended to follow the above mentioned framework (based on the denotational semantics) to arrive at a correct abstract interpreter.

2.2 Output from Z1: A Collecting Analyzer

Z1 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* [HY91].

The input to the generated analyzer is the source program to be compiled. The output from the analyzer is the collecting analysis result.

2.2.1 Program Point

The program points of a source 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.

Example 1 For an assignment statement "`x = b + c;`" its abstract syntax tree might be

```
(= (x) (+ (b) (c)))
```

where (root l-subtree r-subtree). Each node is a program point. Thus, for example, node $\boxed{\texttt{=}}$ is the assignment point, node $\boxed{\texttt{+}}$ is the plus expression point, and so on. □

Example 2 For a sequence of statements "`S1; S2`" its abstract syntax tree might be

```
(begin <S1> <S2>)
```

where `<Si>` is the abstract syntax tree for statement `Si`. Node Pre-state (respectively, post-state) at the program point $\boxed{\texttt{begin}}$ is the program state before the beginning (respectively, after the completion) of the statement sequence. □

2.2.2 Program State

The program state is visible in an interpreter definition. An interpreter is a function that defines, for each language construct, its evaluation rule. The evaluation rule for each language construct 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 the lattices of pre-states and post-states, respectively. The interpreter $\overline{I}$ is usually a recursive function that looks like:

$$
\begin{array}{lll}
\overline{I} = \lambda\sigma.\lambda\overline{x}.\ \mathrm{case}(\sigma)\ \mathrm{of} & & \\
& \sigma \text{ an assignment: } \cdots\overline{I}(\sigma', \overline{x}')\cdots & \text{/* abstract evaluation rule for assignment */} \\
& \sigma \text{ a procedure call: } \cdots\overline{I}(\sigma'', \overline{x}'')\cdots & \text{/* abstract evaluation rule for procedure call */} \\
& \ldots & \\
& \sigma \text{ a constant: } \cdots\overline{I}(\sigma''', \overline{x}''')\cdots & \text{/* abstract evaluation rule for constant */}
\end{array}
$$

2.2.3 Collecting Analysis Computation

System Z1 creates a C code that computes, from an input program, a collecting analysis.

Definition 1 (Collecting Analysis) A collecting analysis associates with each program point the pre-state and the post-state which occur at that point during execution.

How can an abstract interpreter definition be a specification of a collecting analysis? In the previous two sections we clarified what we mean by program points and program states. Program points are the abstract syntax nodes of a program and program states are the value spaces over which an abstract interpreter operates.

For an abstract interpreter $\overline{I}$, let $F_{\overline{I}}$ be the associated functional [1] of the recursive function $\overline{I}$:

$$\begin{aligned} F_{\overline{I}} = \lambda \overline{I}.\lambda\sigma.\lambda\overline{x}.\ &\text{case}(\sigma)\text{ of} \\ &\sigma \text{ an assignment: } \cdots \overline{I}(\sigma', \overline{x}') \cdots \\ &\sigma \text{ a procedure call: } \cdots \overline{I}(\sigma'', \overline{x}'') \cdots \\ &\ldots \\ &\sigma \text{ a constant: } \cdots \overline{I}(\sigma''', \overline{x}''') \cdots \end{aligned}$$

Let $\overline{x}_0$ be the starting pre-state of the program P and Σ_P be the set of program points of P.

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

$$\mathit{Tabulate}(F_{\overline{I}}, \Sigma_P, \overline{x}_0) \quad \text{(See Figure 2.1).}$$

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.

Initially, T_X and T_Y have bottom for each program point $\in \Sigma_P$, except for the program entry point σ_0, whose pre-state value $T_X(\sigma_0)$ is initialized with the starting pre-state x_0. At each iteration, a new post-state $T_Y(\sigma)$ is computed for every program point σ. The input to this computation is the pre-state value $T_X(\sigma)$ that has been mostly recently recorded. The computation is defined by the evaluation rule $F_{\overline{I}}(f)(\sigma)$. When this requires the evaluation of a subexpression σ' of σ, a new pre-state x', which is delivered as an input to σ' by this computation, is joined with the previous pre-state at σ'. This is done by f, which is applied to the subexpression point σ' and the pre-state x'. Note that $F_{\overline{I}}(f)$ replaces the recursive

[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                                 /* evaluation of σ' is required for its parent expression */
    if x' ⋢ T_X(σ') then
        T_X(σ') = T_X(σ') ⊔ x';       /* join the pre-state */
    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;
    ∀σ ∈ Σ_P : 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 the post-state */
    until (T_X ⊑ T'_X) ∧ (T_Y ⊑ T'_Y)              /* repeat until no movement */
end
```

Figure 2.1: Collecting Analysis from an Abstract Interpreter Functional F

calls of $\overline{I}$ by f. Because of the monotonicity of $F_{\overline{I}}(f)(\sigma)$, every iteration deposits non-decreasing pre-state (respectively, post-state) value in T_X (respectively, T_Y). The iteration stops when there is no movement of the state values.

There are two important things about our collecting analysis computation method:

- By means of a separate fixpoint computation driver (*Tabulate*), an abstract interpreter definition, which includes neither a fixpoint operator nor a property-collecting mechanism, becomes a collecting analysis specification.
- The interprocedural control flow graph of the program is constructed dynamically, as the *Tabulate* procedure proceeds.

Complexity of the Collecting Analysis

The worst case complexity of the collecting analysis (Figure 2.1) is proportional to the number of iterations multiplied by the number $|\Sigma_P|$ of the program points of the input program P. The number of iterations is bound by the sum $h(X) + h(Y)$ of the heights of the pre-state lattice X and the post-state lattice Y. This is because at every iteration state value for each program point is moved up along a chain in the corresponding lattice by the join operation until it becomes stable. At each iteration, the semantic evaluation $F(f, \sigma, T_X(\sigma))$ and the comparison $(T_X \sqsubseteq T'_X) \wedge (T_Y \sqsubseteq T'_Y)$ require non-trivial computation. Let C be the complexity for evaluating $F(f, \sigma, T_X(\sigma))$ and $j(X)$ and $j(Y)$ for comparing two elements of X and Y, respectively. The worst case complexity of the collecting analysis becomes

$$\mathcal{O}(|\Sigma_P|(h(X) + h(Y))(C + j(X) + j(Y)))$$

It is noteworthy that, inside Z1, two complexity functions $h()$ and $j()$ does not grow exponentially. Height $h()$ of each lattice that can be defined in Z1 and the complexity $j()$ of the comparison operation are polynomial functions. We will discuss these properties in Chapter 3.

Worklist Algorithm for Computing the Fixpoint

Figure 2.1 shows a basic algorithm to compute the collecting analysis. In reality, system Z1 uses a worklist algorithm which invokes F only for a subset of program points whose T_X and T_Y entries were changed by the previous iteration. The worklist algorithm [Har92] is in Figure 2.2.

Note that we can further improve the speed of this worklist algorithm. It depends on how the **Select** and **Add** procedures are implemented. These determine the order in which program points are selected from and added to the worklist. Depending on this order, the performance of the worklist algorithm will vary. Instead of using a naive approach (e.g.,

```
W: 2^Σ                              /* worklist: pgm points to be evaluated */
w: Σ                                /* a pgm point selected from the worklist */

/* f is called when the evaluation of w (current expression point) requires
   the evaluation of σ (sub expression point of w). */
f(σ: Σ, x: X̄): Ȳ
begin
     record that evaluation of w requires that of σ;
     if (σ was not visited before or x ⋢ T_X(σ)) then
          T_X(σ) = T_X(σ) ⊔ x;      /* new pre-state */
          W = Add(W, σ);            /* put σ into the worklist */
     endif
     return T_Y(σ);
end

Tabulate(F: (Σ → X̄ → Ȳ) → Σ → X̄ → Ȳ, Σ_P: 2^Σ, x_0: X̄): void
T_X: Σ_P → X̄;                       /* program point to pre-state */
T_Y: Σ_P → Ȳ;                       /* program point to post-state */
y: Ȳ
begin
     ∀σ ∈ Σ_P : T_X(σ) = ⊥_X̄;
     ∀σ ∈ Σ_P : T_Y(σ) = ⊥_Ȳ;
     T_X(σ_0) = x_0;                /* initial state at the pgm entry point */
     W = {σ_0}                      /* initialize the worklist with the pgm entry point */
     repeat
          w = Select(W)             /* current expr point to be evaluated */
          y = F(f, w, T_X(w))       /* evaluate */
          if y ⋢ T_Y(w) then
               T_Y(w) = y
               ∀expression point w' whose evaluation needs that of w
                    W = Add(W, w')  /* put it into the worklist */
          endif
     until W ≠ {}
end
```

Figure 2.2: Worklist Algorithm for Collecting Analysis from an Abstract Interpreter Functional F

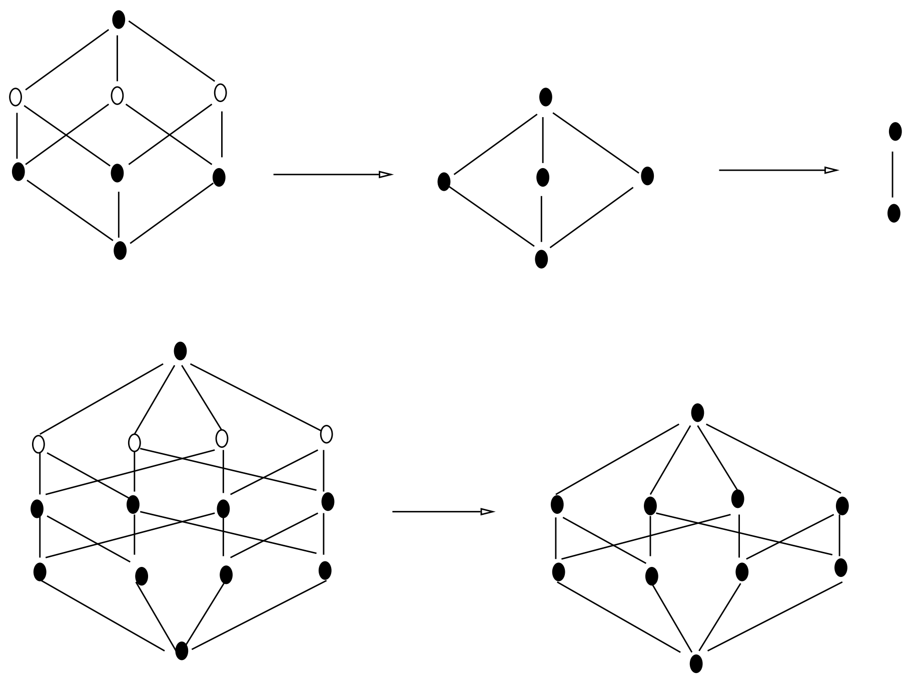


Figure 2.3: Projection Examples

a stack or a queue) if we incorporate the information about the evaluation dependence graph (expression x "depends" on y if the evaluation of x requires that of y) we can remove unnecessary evaluations. We do not address this issue in this thesis. A number of good heuristics for the order are reported in [CH92].

Entailment Graph: A By-product of the Worklist Algorithm

The principal outputs of the collecting analysis are the two tables T_X and T_Y that map each program point to the pre-state and the post-state that occur before and after the program point during execution. An important by-product of the worklist algorithm (Figure 2.2) is the *entailment graph*[Har92] of the input program. The nodes of this graph are the program points. An edge $x \to y$ from a program point x to y indicates that the evaluation of x requires (entails) that of y. When we focus on the edges whose sinks are the entries to procedures, the entailment graph can be seen as the call graph.

2.3 Feedback to Z1: Projection Expressions

After Z1 generated a program analyzer from an input specification, suppose the designer found that the analysis is too costly. An obvious way to design a new one is to rewrite the specification. Below we propose a simpler mechanism (called *projection expressions*) by which the designer may elect to trade accuracy for reduced cost, in order to arrive at a cost-effect analysis. Projections reduce the size of the domains over which an existing abstract interpreter evaluates the input programs. They allow an existing abstract interpreter to become a simpler one which yields a less expensive but less accurate analysis without altering the domain definitions and the interpreter function.

The intuition behind this is as follows. The abstract domains are the value spaces over which the abstract interpretation is computed. The analysis cost and accuracy is largely determined by the structures of the abstract domains (lattices). Analysis cost is determined by the lattice height (the length of its longest chain). It is because the lattice height is the upper bound of the number of evaluation steps at each program point to reach to a stable (no more increasing) value. Analysis accuracy too is affected by the number of elements in a chain. The more elements a chain has, the finer increment of concrete values, when we move up along the chain, its elements may represent. This implies that the analysis result will represent a tighter superset of the input program's run-time behavior than in the case where a smaller number of elements exist in the chain.

A lattice projection is defined as:

Definition 2 A map $\theta: D \to D$ for a lattice D is a *projection* if θ is monotonic ($x \sqsubseteq y$ then $f(x) \sqsubseteq f(y)$), idempotent ($\theta = \theta \circ \theta$), and extensive ($Id \sqsubseteq \theta$).

A projection is a function from a lattice into itself such that its image is still a lattice [GKK+80] but has shorter chains than the original lattice does (this is implied by its extensivity $Id \sqsubseteq \theta$). Figure 2.3 shows how projections simplify the domain structures. When the interpretation function operates over the projected lattice, some elements in a chain, which existed in the original lattice, will be "skipped;" they are not processed during the analysis. Instead, an element into which the projection function project those elements is used. Because the projection is monotonic and extensive, this results in an analysis that is no more costly than the original, and likely less accurate. The upper bound of the number of iterations (Figure 2.1) is reduced in proportion to the reduced height of the projected lattice. By the same token, the new analysis computes no more accurate information than the original one does, because no lower elements are selected by projections for those that are computed by the original analysis. Projection is a simple notion by which we can derive a related (in terms of cost and accuracy) abstract interpreter from an existing one.

Analysis management by projections suggests the following approach to designing and experimenting with an analysis. The user defines one detailed abstract interpreter at the outset. Later, when he wants to trim the analysis cost by trading away accuracy, he defines projections over selected domains. The system takes care of the remaining things so that the generated analyzer conducts an analysis which will vary in cost and accuracy according to the projections.

2.3.1 Limitation of Projections

Lattice projection is a very high-level specification method to control the accuracy and cost of an abstract interpreter. It concerns only the types (lattices) of an abstract interpreter, not the details of the interpretation function.

As we discussed in the previous section, an abstraction interpreter with a set of projections is always related to the interpreter without the projections. Projection is sound in this sense: a projected analysis is related to its original one. Is it also complete? Can any pair of related abstract interpreters be explained as the one being a projected image of the other? Lattice projection would be complete if, whenever two abstract interpreters are related in terms of their cost and accuracy, we were able to find a set of lattice projections which makes one the other. Unfortunately, however, lattice projection is not complete. In other words, not every pair of related abstract interpreters can be explained by lattice projections. That is, there exists a pair of related abstract interpreters, one of which is less accurate (and costs less) than the other, but it cannot be derived by applying projections to the lattices of the other abstract interpreter.

Example 3 As an example, suppose that an abstract interpreter $\overline{I}$ computes a constant value x, and that $\overline{J}$ computes either v or w. In other words, the collecting analysis from $\overline{I}$ computes, for a program P, a constant table that maps every its program point to x, while that from $\overline{J}$ computes, for the same program, a table that maps either to v or to w. In addition, suppose that v and w are related to x ($x \sqsubseteq v$ and $x \sqsubseteq w$). This means $\overline{J}$ is a less accurate analysis than $\overline{I}$. However, there exist no projection that maps x sometimes to v and sometimes to w (a one-to-many map!). The best projection we can think of is one that maps x into the least upper bound of v and w. This means that there exists no projection that makes $\overline{I}$ become $\overline{J}$; any projection projects the lattice into one that renders strictly higher analysis result than $\overline{J}$.$\square$

The incompleteness of projection implies that there may be times when to obtain a desired analysis we will have to modify the underlying domains and semantics of an abstract interpreter.

However, many interesting analyses with acceptable accuracy-cost behavior can be derived by applying various projections to an existing abstract interpreter. Chapter 5 includes a series of experiments which support this claim.

Chapter 3

Domains and Projections in Z1

In the previous chapter we presented the formalism for abstract interpretation that is supported by system Z1, and gave an overview of how an abstract interpreter definition was processed to generate a collecting analyzer. We also discussed that the complexity of a generated collecting analyzer is proportional to two measures of the abstract domains: their heights and the complexity of the partial order operation.

In this chapter we will define precisely a space of abstract domains that is supported by system Z1. This space is called *finite modular atomic lattices.* Every lattice that can be defined in Z1 as an abstract domain is a finite modular atomic lattice. Finite modular atomic lattices have several important characteristics:

- Lattice height is a polynomial function.
- Comparison operation ($x \sqsubseteq y$) requires a polynomial time.
- Expressing the projections over a lattice is made easy.

The first two properties are important because the complexity of a generated collecting analysis is determined by them. These two properties result from the finiteness and atomicity of the lattices. The last property comes from the modularity.

3.1 Domains in Z1

Domains in abstract interpreters are the abstract value spaces over which the abstract interpretation is computed. In Section 2.1.2 we defined what each abstract value means. Each abstract domain is structured into a finite-height lattice, whose elements denote ideals of the corresponding domain of the concrete, instrumented semantics (which is written in the denotational semantics formalism). The partial order between the elements is the set

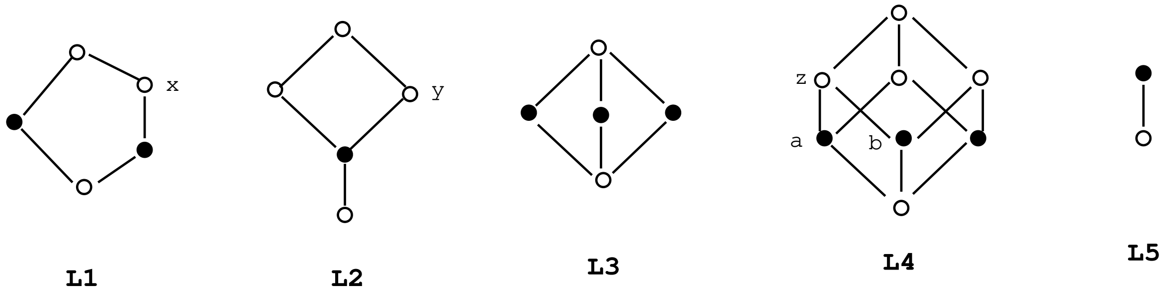


Figure 3.1: Lattices

inclusion of ideals that the abstract elements denote. Note that that the ordering of x and y ($x \sqsubseteq y$) indicates that y denotes a larger ideal (a larger but less accurate value) than x. In Z1, it is single "token" that an abstract interpreter processes during analysis for each abstract value. (This token is in fact the address of the representation for the abstract value.) An ideal that an abstract value denotes is not represented in the computer; this resides only inside the interpreter designer's mind. He knows the association between an abstract value (a representation produced during analysis) and what the token means (an *ideal* of a concrete domain).

Inside Z1 the user can define only a *finite modular atomic lattice.* The space $\mathcal{L}$ of finite modular atomic lattices that is supported by Z1 will be defined precisely below.

3.1.1 Preliminaries

Let us begin with some definitions.

Definition 3 (Lattice) A lattice is a partial order set such that any of its two elements x, y have a least upper bound ($x \sqcup y$) and a greatest lower bound ($x \sqcap y$).

Some lattices are in Figure 3.1. We will refer to these lattices from time to time.

Definition 4 (Bottom and Top) Bottom $\perp$ (respectively, top $\top$) of a lattice, if it exists, is the element that is lower (respectively, higher) than all other elements.

Note that bottom and top are unique if they exist.

Definition 5 (Sublattice) A lattice S is a sublattice of L if and only if $S \subseteq L$ and for any $x, y \in S$, $x \sqcup y$ ($\in L$) and $x \sqcap y$ ($\in L$) are also in S.

Definition 6 (Chain) A subset S of a lattice L is a chain iff every two elements $x, y \in S$ are related ($x \sqsubseteq y$ or $y \sqsubseteq x$).

A finite chain always has a least element that is lower than all other elements in the chain. Likewise, it has a greatest element. When we say "a chain between x and y" it means a chain whose least and greatest elements are x and y, respectively.

A chain between two elements is *maximal* if it cannot be expanded to include any more elements. Note that more than one maximal chain can exist between two elements.

Definition 7 (Lattice Isomorphism) Two lattices L and M are isomorphic if and only if there exists a one-to-one and onto correspondence $\phi\colon L \to M$ such that

$$x \sqsubseteq y \iff \phi(x) \sqsubseteq \phi(y) \quad \forall x, y \in L. \qquad \text{(order preserving)}$$

Usually in the literature, a lattice isomorphism ϕ is defined as an one-to-one and onto map which is both a join and meet homomorphism ($x \sqcup y = \phi(x) \sqcup \phi(y)$, similarly for $\sqcap$). Our definition follows from the fact that a lattice isomorphism is equivalent to an order isomorphism (an order preserving one-to-one and onto map; see Definition 1.11 and Proposition 5.11 in [BH91]). Our definition helps to simplify the proof of Property 8 (page 39) because it does not involve the join and meet operations.

Definition 8 (Modular Lattice) A lattice L which satisfies the condition

$$z \sqsubseteq x \Longrightarrow x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup z$$

for all $x, y, z \in L$ is a *modular* lattice.

An alternative definition and important characteristic of a modular lattice is:

Definition 9 (Modular Lattice) A lattice is modular if and only if it contains no sublattice isomorphic to the pentagonal lattice **L1** (Figure 3.1).

Given this characteristics it is not surprising that in a modular lattice any two maximal chains between two elements, if they are finite, are of the same length. This is **Jordan-Hölder** theorem [Rut65].

Definition 10 (Atom) A non-bottom element x of a lattice L which satisfies

$$y \sqsubseteq x \Longrightarrow y = \bot \text{ or } x = y \qquad \forall y \in L$$

is an atom. We write $Atoms(L)$ to indicate the set of atoms of a lattice L.

An atom is an element which is immediately above bottom. Note that, by definition, all atoms are incomparable.

Example 4 For those lattices in Figure 3.1, black dots are the atoms. For the two-element lattice **L5**, $\top$ is also an atom. □

Definition 11 (Atomic Lattice) A lattice is atomic if and only if every non-bottom element is the join of some atoms.

Definition 12 (Representation Set) For an element x of an atomic lattice, its representation set $Rep(x)$ is defined as

$$Rep(x) = \begin{cases} \{\perp\} & x = \perp; \\ \{x\} & x \text{ is an atom;} \\ \text{maximal set of atoms whose join is } x & \text{otherwise.} \end{cases}$$

Example 5 For lattices in Figure 3.1, **L1** and **L2** are not atomic lattices. In **L1** (respectively, **L2**), x (respectively, y) is not a join of any atoms. **L3**, **L4**, and **L5** are atomic. $Rep(\top \in \mathbf{L3}) = Atoms(\mathbf{L3})$ and $Rep(z \in \mathbf{L4}) = \{a, b\}$. □

Property 1 *An atomic lattice is finite if and only if its atom set is finite.*

Proof. Every non-bottom element of an atomic lattice is the join of some atoms. If the atom set is finite, its subsets are of finite number, which is the maximal number of non-bottom elements.
□

We shall define two primitive lattices: a "flat lattice" and a "powerset lattice." They are primitive because they are constructed from sets (rather than from other lattices). A compound lattice is one constructed from other lattices. A compound lattice is either a "product lattice" or an "atomic function lattice." We shall give their definitions and derive properties of this family of lattices.

3.1.2 Primitive Lattices

Definition 13 (Flat Lattice) A lattice L which satisfies

$$x \sqsubseteq y \Longrightarrow \text{ either } x = y, x = \perp, \text{or } y = \top \qquad \forall x, y \in L$$

is a flat lattice. A flat lattice consists only of bottom, top, and atoms. We write $S_\perp^\top$ to indicate a flat lattice whose atom set is S.

Example 6 **L3** and **L5** in Figure 3.1 are flat lattices.□

Property 2 *A flat lattice $S_\perp^\top$ is modular.*

Proof. Since it has no sublattice isomorphic to the pentagonal lattice **L1** (see Figure 3.1).
□

Property 3 *A flat lattice $S_\bot^\top$ is atomic.*

Proof. There is single non-atom element: $\top$, whose representation set $Rep(\top)$ is the set of atoms S.
□

Definition 14 (Powerset Lattice) A powerset lattice is a powerset of a set with set inclusion as its partial order. We write 2^S to indicate the powerset lattice from a set S.

$$\begin{array}{lcl} 2^S & = & \{\, x \mid x \subseteq S \,\} \\ x \sqsubseteq y & = & x \subseteq y \\ x \sqcup y & = & x \cup y \\ x \sqcap y & = & x \cap y \end{array}$$

Example 7 For set $S = \{1, 2, 3\}$,

$$2^S = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

which is **L4** in Figure 3.1. □

Property 4 *Powerset lattice is modular.*

Proof. When we replace the partial order by subset relation, and the join (respectively, meet) by set union (respectively, intersection), we can see that the postulate in Definition 8 holds:

$$\begin{array}{rcll} x \cap (y \cup z) & = & (x \cap y) \cup (x \cap z) & \text{(DeMorgan's rule)} \\ & = & (x \cap y) \cup z. & (\text{if } z \subseteq x) \end{array}$$

□

Property 5 *Powerset lattice is atomic.*

Proof. The atoms of a powerset lattice 2^S are the set of singleton subsets of S: $Atoms(2^S) = \{\{a\}\}_{a \in S}$. A non-atom element is a subset of S, which is the join (set union) of some atoms. $Rep(x \in 2^S) = \cup_{a \in x}\{\{a\}\}$.
□

So far, we defined flat lattice $S_\bot^\top$ and powerset lattice 2^S and showed that these lattices are modular and atomic. Also, if the base set S is finite the lattice is also finite. We call these two lattices "primitive" because they are not constructed from other lattices, which we call "compound" lattices. We shall define two compound lattices: product lattice and atomic function lattice.

3.1.3 Compound Lattices

Definition 15 (Product Lattice) A product lattice $A \times \cdots \times B$ from lattices $A, \ldots, B$ is their Cartesian product with component-wise order:

$$\begin{array}{lcl} A \times \cdots \times B & = & \{ \langle a, \ldots, b \rangle \mid a \in A, \ldots, b \in B \} \\ \langle a, \ldots, b \rangle \sqsubseteq \langle a', \ldots, b' \rangle & = & a \sqsubseteq a', \cdots, \text{ and } b \sqsubseteq b' \\ \langle a, \ldots, b \rangle \sqcup \langle a', \ldots, b' \rangle & = & \langle a \sqcup a', \ldots, b \sqcup b' \rangle \\ \text{Similary for } \sqcap. & & \end{array}$$

Property 6 *Product lattice is modular if its component lattices are modular.*

Proof. Without loss of generality, consider a product lattice $A \times B$ of two modular lattices A and B. Let $\langle a, b \rangle, \langle v, w \rangle, \langle m, n \rangle \in A \times B$. That $\langle m, n \rangle \sqsubseteq \langle a, b \rangle$ means $m \sqsubseteq a$ and $n \sqsubseteq b$, which means, since A and B are modular,

$$\begin{array}{c} a \sqcap (v \sqcup m) = (a \sqcap v) \sqcup m \\ b \sqcap (w \sqcup n) = (b \sqcap w) \sqcup n. \end{array}$$

Thus

$$\begin{array}{rcl} \langle a, b \rangle \sqcap (\langle v, w \rangle \sqcup \langle m, n \rangle) & = & \langle a \sqcap (v \sqcup m), b \sqcap (w \sqcup n) \rangle \\ & = & \langle (a \sqcap v) \sqcup m, (b \sqcap w) \sqcup n \rangle \\ & = & (\langle a, b \rangle \sqcap \langle v, w \rangle) \sqcup \langle m, n \rangle. \end{array}$$

□

Property 7 *Product lattice is atomic if its component lattices are atomic.*

Proof. Without loss of generality consider a product lattice $A \times B$ of two atomic lattices A and B. Let $\langle a, b \rangle$ be a non-bottom element of it. Since the component lattices are atomic,

$$\begin{array}{rcl} \langle a, b \rangle & = & \langle \sqcup Rep(a), \sqcup Rep(b) \rangle \\ & = & \langle \sqcup Rep(a), \bot \rangle \sqcup \langle \bot, \sqcup Rep(b) \rangle \\ & = & (\bigsqcup_{\alpha \in Rep(a)} \langle \alpha, \bot \rangle) \sqcup (\bigsqcup_{\beta \in Rep(b)} \langle \bot, \beta \rangle), \\ & & \text{which are the join of some atoms of } A \times B. \end{array}$$

Note that

$$Atoms(A \times \cdots \times B) = \{ \langle a_i, \cdots, \bot \rangle \mid a_i \in Atoms(A) \} \cup \cdots \cup \{ \langle \bot, \cdots, b_i \rangle \mid b_i \in Atoms(B) \}.$$

□

Another compound lattice is an *atomic function lattice.*

Definition 16 (Atomic Function Lattice) An atomic function lattice $A \twoheadrightarrow B$ is a set of functions f from an atomic lattice A to a lattice B such that

- f is strict: $f(\bot) = \bot$.
- f is atomic: every image $f(x)$ for $x \in A$ is the join of the images $f(x_i)$ over x_is that represent x (i.e., $Rep(x) = \{x_i\}$).

The partial order, join, and meet operation is as follows:

$$
\begin{array}{lcl}
A \twoheadrightarrow B & = & \{\, f\colon A \to B \mid f(\bot) = \bot \text{ (strict) and } f(x) = \sqcup_{x_i \in Rep(x)} f(x_i) \text{ (atomic)} \} \\
f \sqsubseteq g & = & f(x) \sqsubseteq g(x) \forall x \in A \\
f \sqcup g & = & \lambda x. f(x) \sqcup g(x) \qquad (*) \\
f \sqcap g & = & Atomize(\lambda x. f(x) \sqcap g(x))
\end{array}
$$

For a function $f\colon A \to B$ ($A \to B$, not $A \twoheadrightarrow B$, is the point-wise ordered function lattice) $Atomize(f)$ is the greatest atomic function which is lower than f.

We must assure that the point-wise definition $(*)$ of $\sqcup$ is well-defined. In other words, for two strict and atomic functions f and g, their join $\lambda x. f(x) \sqcup g(x)$ must still be strict and atomic. It is clearly strict. It is atomic because

$$
\begin{array}{lcll}
\lambda x. f(x) \sqcup g(x) & = & \lambda x. (\bigsqcup_{x_i \in Rep(x)} f(x_i)) \sqcup (\bigsqcup_{x_i \in Rep(x)} g(x_i)) & (f \text{ and } g \text{ are atomic.}) \\
 & = & \lambda x. \bigsqcup_{x_i \in Rep(x)} (f(x_i) \sqcup g(x_i)). & (\sqcup \text{ is associative.})
\end{array}
$$

Actually,

Property 8 *The atomic function lattice $A \twoheadrightarrow B$ is isomorphic to the product lattice*

$$\underbrace{B \times \cdots \times B}_{n} \quad \textit{(we will write } B^n \textit{ for this)}$$

where n is the number of atoms $|Atoms(A)|$ of A.

Proof. We prove this by showing a lattice isomorphism ϕ (Definition 7) from $A \twoheadrightarrow B$ to B^n where $n = |Atoms(A)|$. Define

$$\phi(f) = \langle f(x_1), \ldots, f(x_n) \rangle, \quad \text{with } \{x_i\} = Atoms(A).$$

• ϕ is one-to-one and onto: $f \neq g$ implies that there exists an atom $x_i \in Atoms(A)$ such that $f(x_i) \neq g(x_i)$, which implies $\phi(f) \neq \phi(g)$. Given $y = \langle y_1, \cdots, y_n \rangle \in B^n$, let

$$f = \lambda x. \begin{cases} \bot & x = \bot; \\ y_i & x \in Atoms(A); \\ \sqcup_{x_i \in Rep(x)} y_i & \text{otherwise.} \end{cases}$$

Clearly, f is strict and distributive and $\phi(f) = y$.

- ϕ is order isomorphic ($x \sqsubseteq y \iff \phi(x) \sqsubseteq \phi(y)$): $f \sqsubseteq g$ implies that $f(x) \sqsubseteq g(x)$ for all $x \in A$, which implies that $f(x_i) \sqsubseteq g(x_i)$ for every atom x_i of A. Thus $\phi(f) \sqsubseteq \phi(g)$. Conversely, suppose $\phi(f) \sqsubseteq \phi(g)$. It implies that $f(x_i) \sqsubseteq g(x_i)$ for every atoms x_i of A. By definition of atomic functions ($\forall x \in D_1 : f(x) = \sqcup_{x_i \in Rep(x_i)} f(x_i)$), this in turn implies that $f(x) \sqsubseteq g(x)$ for all $x \in D_1$. □

Property 9 *Atomic function lattice $A \twoheadrightarrow B$ is modular if B is modular.*

Proof. Because $A \twoheadrightarrow B$ is isomorphic to the product lattice $B^{|Atoms(A)|}$ and because a product lattice is modular if its component lattices (in this case B) are modular (Property 6).
□

Property 10 *Atomic function lattice $A \twoheadrightarrow B$ is atomic if B is atomic.*

Proof. Because $A \twoheadrightarrow B$ is isomorphic to the product lattice $B^{|Atoms(A)|}$ and because a product lattice is atomic if its component lattices (in this case B) are atomic (Property 7).

Note that

$$Atoms(A \twoheadrightarrow B) = \{\, \bot[b_i/a_1] \mid b_i \in Atoms(B)\,\} \cup \cdots \cup \{\, \bot[b_i/a_n] \mid b_i \in Atoms(B)\,\}$$

where $\{a_1, \cdots, a_n\} = Atoms(A)$.
□

3.1.4 The Lattice Space $\mathcal{L}$

Definition 17 ($\mathcal{L}$) Define $\mathcal{L}$ be the set of lattices, which is constructed by two rules:

- When S is a finite set, $S_\bot^\top$ (flat lattice) and 2^S (powerset lattice) are members of $\mathcal{L}$.
- When $A, \cdots, B$ are members of $\mathcal{L}$, $A \times \cdots \times B$ (product lattice) and $A \twoheadrightarrow B$ (atomic function lattice) are members of $\mathcal{L}$.

Property 11 *Every lattice in $\mathcal{L}$ is finite modular atomic lattice.*

Proof. Every flat or powerset lattice is finite because its base set is finite. It is also modular and atomic (Property 2,3,4,and 5). Inductively, every product lattice is also finite, modular, and atomic (Property 6 and 7). So is every atomic function lattice (Property 9 and 10).
□

Note that the construction of an atomic function lattice $A \twoheadrightarrow B$ can be done freely since any lattice A in $\mathcal{L}$ is atomic.

One important implication by the modularity property is:

Property 12 *For each lattice in $\mathcal{L}$, every maximal chain between two elements is of the same length.*

Proof. In general, more than one maximal chain can exist between two elements. However, because every lattice in $\mathcal{L}$ is finite and modular, every maximal chain between two elements is of the same length (Jordan-Hölder theorem [Rut65]). $\square$

Thank to this, every element of L in $\mathcal{L}$ has unique height and depth.

Definition 18 (Element Height (Depth)) The height $h(x)$ (respectively, the depth $d(x)$) of $x \in L \in \mathcal{L}$ is the length of a maximal chain from $\perp$ (respectively, $\top$) to x. Note that there can exist more than one maximal chain, but all are of the same length (Property 12).

The length of a chain is the number of elements in the chain minus one.

We use the heights and depths of lattice elements in the projection expressions.

Property 13 *For each lattice in $\mathcal{L}$, an element's height (respectively depth) can be computed inductively on the lattice 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 $f \in A \twoheadrightarrow B$, $h(f) = h(f(x_1)) + \cdots + h(f(x_n))$ with $\{x_i\} = Atoms(A)$, since $A \twoheadrightarrow B$ is isomorphic to B^n with $n = |Atoms(A)|$ (Property 8).
$\square$

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

Because the height of a lattice is defined by the height of its top element, it can easily be seen that the height $h(L)$ of a lattice $L \in \mathcal{L}$ is expressed in a polynomial form:

$$h(L) = \begin{cases} 2 & L = S_{\perp}^{\top}; \\ |S| & L = 2^{S}; \\ h(A) + \cdots + h(B) & L = A \times \cdots \times B; \\ h(B) \times |Atoms(A)| & L = A \twoheadrightarrow B. \end{cases}$$

(Note that the number of atoms $|Atoms(L)|$ of a lattice $L \in \mathcal{L}$ is also a polynomial form.) This is important, because a lattice height is one factor that determines the complexity of a collecting analysis generated by Z1. Another factor is the complexity $j(L \in \mathcal{L})$ of the partial order operation $x \sqsubseteq y$. It can be similarly seen that the complexity is expressed as a polynomial form:

$$j(L) = \begin{cases} 1 & L = S_{\perp}^{\top}; \\ |S|^2 & L = 2^{S};\text{ set inclusion} \\ j(A) + \cdots + j(B) & L = A \times \cdots \times B; \\ j(B) \times |Atoms(A)| & L = A \twoheadrightarrow B. \end{cases}$$

As you might see, the join operation also requires a polynomial time computation.

3.1.5 Domains in Z1 Are Elements of $\mathcal{L}$

- System Z1 provides some constructs to define finite sets: the user can define an enumerated set, a range set, an index set, a product set, and a union set, all of which are finite.
- System Z1 provides the constructs ($\cdot_{\perp}^{\top}, 2^{\cdot}, \times, \twoheadrightarrow$) by which the user can define, from a defined set, a flat lattice or a powerset lattice and, from already defined lattices, a product lattice or an atomic function lattice.

Therefore, every lattice definable in Z1 is a member of $\mathcal{L}$, and every lattice in $\mathcal{L}$ is definable in Z1 if its underlying set is definable in Z1.

3.2 Projections in Z1

In Section 2.3 we define lattice projections that are supported in Z1. As we discussed in Section 2.3, projections control the cost vs accuracy tradeoff of an abstract interpreter.

Definition 19 (Projection) A map $\theta: D \to D$ is a projection if θ is monotonic, idempotent ($\theta = \theta \circ \theta$), and extensive ($Id \sqsubseteq \theta$).

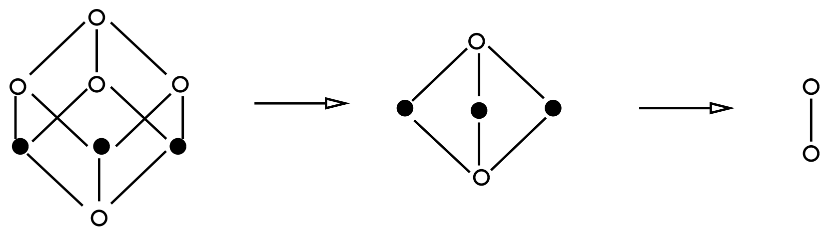


Figure 3.2: Lattice Projections

Example 8 Let D be a powerset lattice 2^S of a set S. We can project it into several simplified lattices. 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

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

This is the first projection in Figure 3.2. □

Example 9 Suppose we have a flat lattice $S_\perp^\top$. We can nullify the evaluation effects over this lattice by projecting all of its atoms into top. This is the second projection in Figure 3.2. □

Example 10 Suppose we have a product lattice $A \times B$. Suppose we find all the pairs $\langle \top_A, \cdot \rangle$ are no more useful than $\top = \langle \top_A, \top_B \rangle$. Then, we can use a projection θ to avoid all such elements during analysis:

$$\theta = \lambda \langle a, b \rangle . \begin{cases} \top, & \text{if } a = \top_A; \\ \langle a, b \rangle, & \text{otherwise.} \end{cases}$$

□

Example 11 Suppose we have an atomic function lattice $A \twoheadrightarrow B$. Suppose we find that every atomic function whose entry for some atom satisfies a condition δ is no more useful than $\top$. Then, we can use a projection θ to avoid all such elements during analysis:

$$\theta = \lambda f \in A \twoheadrightarrow B. \begin{cases} \top, & \text{if } \exists a \in \mathit{Atoms}(A) : \delta(f(a)) = 1; \\ f, & \text{otherwise.} \end{cases}$$

In order that the above projection be monotonic, the condition δ must satisfy

$$\delta(x) = 1 \Longrightarrow \delta(y) = 1 \quad \forall x \sqsubseteq y.$$

□

How can we compute an analysis over the projected domains? Suppose we have projections θ_A and θ_B for lattices A and B. For a monotonic operation $f\colon A \to B$, we wrap f with those projections: $\theta_B \circ f \circ \theta_A$.

Property 14 *For a monotonic operation $f\colon A \to B$ and projections $\theta_A\colon A \to A$ and $\theta_B\colon B \to B$,*

$$f \sqsubseteq \theta_B \circ f \circ \theta_A.$$

Proof.

$$\begin{array}{rcll} x & \sqsubseteq & \theta_A(x) & (Id \sqsubseteq \theta_A) \\ f(x) & \sqsubseteq & (f \circ \theta_A)(x) & (f \text{ is monotonic}) \\ (\theta_B \circ f)(x) & \sqsubseteq & (\theta_B \circ f \circ \theta_A)(x) & (\theta_B \text{ is monotonic}) \end{array}$$

□

Therefore, every semantic operation f wrapped with projections of the input/output lattices guarantees that the semantic operation be evaluated over the projected lattices and that its result is a safe one. Because of the idempotency of projections, it's enough to apply each projection only once (per application of f).

The above property indicates that projections allow us to tune an analysis *without changing any semantic operation.* Projections are specified for some lattices, independently of the semantic operations over those lattices. Analysis over the projected lattices is done by applying corresponding projections before and after an operation is applied to a lattice element. In reality, Z1 uses a more sophisticated method that insures that only elements in the projected lattice occur, by projecting at the moment that lattice elements are created.

In Z1, limited set of projections can be defined. Every projection in Z1 is a kind of characteristic function:

Definition 20 Every projection $\theta\colon D \to D$ inside Z1 is a function of the following form:

$$\lambda x. \begin{cases} x & \delta_D(x) = \text{false}; \\ \top & \delta_D(x) = \text{true}. \end{cases} \qquad (1)$$

Projection $\theta_D(x)$ projects x into top if x satisfy the projection predicate $\delta_D(x)$, otherwise, it projects x into itself. Defining a projection θ_D in Z1 is reduced to defining its predicate δ_D.

For (1) to be a projection, (1) should be idempotent, extensive, and monotonic. Clearly, (1) is an idempotent function: $(1) \circ (1) = (1)$ and is extensive: $Id \sqsubseteq (1)$. For (1) to be monotonic, the projection predicate δ_D must be *upward consistent*. We will prove this in Property 15 after we define the upward consistency.

Definition 21 (Upward-Consistency) A predicate $\delta_D: D \rightarrow \{\text{true}, \text{false}\}$ over a lattice D is upward-consistent if and only if

$$\delta_D(x) = \text{true} \Longrightarrow \forall y \sqsupseteq x : \delta_D(y) = \text{true}.$$

Property 15 *If* δ_D *is* upward consistent, *then function (1) is monotonic.*

Proof. Let $x \sqsubseteq y$. Suppose $\theta_D(x) = x$. Since $\delta_D(y)$ is either y or $\top$, $\theta_D(x) \sqsubseteq \theta_D(y)$. Suppose $\theta_D(x) = \top$. This means either $\delta_D(x) = \text{true}$ or $x = \top \wedge \delta_D(x) = \text{false}$. For the first case, by the condition of $delta_D$, $delta_D(y) = \text{true}$, which means $\theta_D(y)$ is also $\top$. For the second case, $y = x = \top$. $\square$

Z1 provides a set of constructs by which the user can define the projection predicates over the lattices. And, every projection predicate in Z1 is upward-consistent. We will prove this in Property 16. Before that, let's first see which constructs are provided by Z1 to define projection predicates. The user can define a projection predicates in seven ways inside Z1:

Definition 22 (Projection Predicates δ in Z1) The projection predicates ($D \rightarrow \{\text{true}, \text{false}\}$) in Z1 can be defined in seven ways:

- $\lambda x.x = \top$ is definable.
- $\lambda x.h(x) > n$ is definable ($h(x)$ is the height of x).
- $\lambda x.d(x) < n$ is definable ($d(x)$ is the depth of x).
- Using the projection predicates $\delta_X, \cdots, \delta_Y$ defined for lattices $X, \ldots, Y$, we can define two kinds of projection predicates for the product lattice $X \times \cdots \times Y$:

  $$\lambda\langle x, \cdots, y\rangle.\delta_X(x) \wedge \cdots \wedge \delta_Y(y)$$

  and

  $$\lambda\langle x, \cdots, y\rangle.\delta_X(x) \vee \cdots \vee \delta_Y(y)$$

- Using the projection predicate δ_B defined for lattice B, we can define two kinds of projection predicates for the atomic function lattice $A \twoheadrightarrow B$:

  $$\lambda f \in A \twoheadrightarrow B.\forall a \in Atoms(A) : \delta_B(f(a))$$

  and

  $$\lambda f \in A \twoheadrightarrow B.\exists a \in Atoms(A) : \delta_B(f(a))$$

Property 16 *Every projection predicate δ in Z1 is upward consistent (Property 15). Therefore, it defines a projection in Z1 (the function (1) of Definition 20).*

Proof. When $x = \top$ every $y \sqsupseteq x$ is also $\top$. When the height (respectively, depth) of x is larger (respectively, smaller) than n, so is every $y \sqsupseteq x$ (Property 13). By induction, suppose $\delta_X, \cdots, \delta_Y$ are upward consistent. Let

$$\langle x, \cdots, y\rangle \sqsubseteq \langle x', \cdots, y'\rangle$$

and let

$$\delta_X(x) \wedge \cdots \wedge \delta_Y(y) = \text{true}.$$

This implies

$$\delta_X(x') \wedge \cdots \wedge \delta_Y(y') = \text{true}$$

because $x \sqsubseteq x' \wedge \cdots \wedge y \sqsubseteq y'$ and every $\delta_\bullet$ is upward consistent. Similarly, let

$$\delta_X(x) \vee \cdots \vee \delta_Y(y) = \text{true}.$$

This implies, some $\delta_\bullet$ has true value for corresponding component value. Because $x \sqsubseteq x' \wedge \cdots \wedge y \sqsubseteq y'$ and $\delta_\bullet$ is upward consistent

$$\delta_X(x') \vee \cdots \vee \delta_Y(y') = \text{true}.$$

Because an atomic function lattice $A \rightarrow B$ is isomorphic to the product lattice $B^{|Atoms(A)|}$, when δ_B is upward consistent,

$$\forall a \in Atoms(A) : \delta_B(f(a)) \quad \text{and} \quad \exists a \in Atoms(A) : \delta_B(f(a))$$

are also upward consistent. □

3.3 Chapter Summary

In this chapter we showed that the space of lattices that can be defined inside Z1 is a closed one where every lattice construction operator generates a *finite modular atomic* lattice. Moreover, the provided lattice constructors can freely be applied to any finite modular atomic lattice.

Because every lattice in Z1 is finite and atomic, a generated analysis has a polynomial time complexity. This is because, for finite atomic lattices, lattice height and partial order operation complexity are polynomial functions. Because of the modularity of every lattice in Z1, we can easily compute the heights of lattice elements for use in projections.

We also showed that every projection predicate that can be defined inside Z1 is *upward consistent* (Definition 21), thus the function that it defines (Definition 20) becomes a projection. The user defines a projection predicate and system Z1 use the function determined by the predicate as a projection.

Chapter 4

The Specification Language Z1

In this chapter, we present the specification language Z1 in which we specify an abstract interpreter. We give the language the same name as the system.

An abstract interpreter consists of abstract domains and abstract evaluation rules. In Z1, every abstract domain is a lattice in $\mathcal{L}$ (Definition 17 in Chapter 3). Abstract evaluation rules in Z1 is a collection of monotonic functions between abstract domains. An abstract interpreter specification in Z1 is compiled into a C program that computes the specified collecting analysis as outlined in Section 2.2.

4.1 Getting Acquainted

The specification language Z1 is

- a strongly-typed language: every expression has a unique type which can be determined statically.
- an eager-evaluation language: every expression is evaluated eagerly.
- supports user-defined types: the user can define types and use their values.
- statically-scoped language: variable binding is lexically determined.

4.1.1 Evaluation Rule

Expressions are evaluated eagerly; that is, before an expression is evaluated, each of its subexpressions is evaluated.

4.1.2 Type Structure

Every expression is uniquely typed, and its type can be determined statically. No implicit type coercion is used. Type casting operations are provided for occasions in which an explicit cast is necessary.

Types in Z1 are of two kinds: *sets* and *domains*. A domain is a *finite-height lattice*. There are two pre-defined sets: `number` (integer) and `syntree`. All other types are defined by the user. The user can define an enumerated set, an integer range set, an index set, a product set, and a union set. All these sets are finite. Similarly, the user can define a flat domain, a powerset domain, a product domain, and a function domain. These domain constructs correspond to the lattice constructs ($^{\top}_{\bot}, 2^{\cdot}, \times,$ and $\rightarrow$) defined in Chapter 3. Every abstract domain in Z1 is, by these constructions, a finite modular atomic lattice in $\mathcal{L}$ (Definition 17 in Chapter 3).

4.1.3 Scope Rule

Every variable in Z1 is statically scoped. The scope of a variable is determined from the lexically innermost binding construct. There are two binding constructs: let expressions and function definitions.

4.1.4 Restrictions

- Z1 does not allow a fixpoint operator to appear inside an abstract interpreter specification. As we discussed in Section 2.2, system Z1 uses a global fixpoint operation which computes program states for each program point.
- Z1 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. Together with monotonic operations over the finite-height lattices, this guarantees the termination of the specified program analysis.

  [Example] A type definition $S = A + B \times S$ is reflexive; `S` is defined in terms of `S` itself. This reflexive definition is not allowed in Z1 since it permits an infinite number of elements in `S`.

4.1.5 Specification Structure

An abstract interpreter specification in Z1 consists of

- type definitions: sets and domains.
- interpretation functions: auxiliary functions and the main interpreter function.

- projection expressions: projecting some domains into simpler ones.

4.2 Notation

To present the language Z1, 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 Z1, 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.

4.3 Types

User-defined types are the specifications for the abstract domains over which the abstract interpreter evaluates the target programs. When we say "two types are same" we mean that they are name-equivalent or in a synonym relation; structural equivalence does not count.

The user defines types in the following hierarchy:

1. Defines primitive sets.
2. Constructs compound sets from primitive sets.
3. Constructs primitive domains from sets.
4. Builds compound domains from primitive domains.

Every domain that can be defined in Z1 is a finite modular atomic lattice. It is finite because the underlying sets are finite. It is modular and atomic because Z1 supports no more than the lattice constructs defined in Chapter 3.

Types are defined by

$$\boxed{\texttt{(types } \textit{typedef}^+\texttt{)}}$$

4.3.1 Sets

A set S can be defined in five ways. All set types must be named uniquely. A set cannot be defined recursively. This means every user-defined set is a finite set.

`(set` S `(elements` $\iota_1 \cdots \iota_n$`))`	enumerated set	$S = \{\iota_1, \ldots, \iota_n\}$,
`(set` S `(range` z_1 z_2`))`	integer range set	$S = \{\, i \in \mathbf{Z} \mid z_1 \leq i \leq z_2 \,\}$,
`(set` S `(index` ρ`))`	index set bounded by $\rho()$	$S = \{\, i \in \mathbf{Z} \mid 0 \leq i \leq \rho() \,\}$,
`(set` S `(*` $S_1 \cdots S_n$`))`	product set	$S = S_1 \times \cdots \times S_n$,
`(set` S `(+` $S_1 \cdots S_n$`))`	union set	$S = S_1 + \cdots + S_n$,
`(set` S' `(synonym` S`))`	an equivalent set except in name.	

- The element names ι of an `elements` set must be distinct.
- A `range` set's z_1 (lower bound) must be less than or equal to z_2 (upper bound).
- The `index` set definition is used for an index set from 0 $\{0, 1, \ldots \rho()\}$ whose upper bound is determined by a target program. ρ must be a name of a C procedure which returns a non-negative integer. $\rho()$ must, of course, returns only one value.

  [Example] To define a set `Id` of program identifiers,

  ```
  (set Id (index NumOfIds))
  ```

  where `NumOfIds()` returns the number of identifiers in a program.
- A product set `(set` S `(*` $S_1 \cdots S_n$`))` is the Cartesian product of S_is. Since Z1 uses the set name as the component selector, the component set names should be distinct. When necessary, `synonym` set definition is used to make two different names for a set.

  [Example] For a product set `(set S (* A A))`, we define

  ```
  (set B (synonym A))
  (set S (* A B))
  ```
- A union set `(set` S `(+` $S_1 \cdots S_n$`))` is the union of S_is.
- A synonym set `(set S (synonym T))` is identical to `T` except in name.

4.3.2 Domains

A domain is a finite-height lattice. Domains are the (abstract) data value spaces over which the abstract evaluation of a target program is conducted. All domains that can be defined in Z1 are finite modular atomic lattices.

All domains are named uniquely. A domain cannot be defined recursively; this, together with the absence of infinite sets, ensures the finite-height of domains.

A domain can be defined in five ways.

`(domain` D `(flat` S`))`	flat domain of a set	$D = S_{\bot}^{\top}$,
`(domain` D `(2^` S`))`	powerset domain of a set	$D = 2^{S}$,
`(domain` D `(*` $D_1 \cdots D_n$`))`	product domain	$D = D_1 \times \cdots \times D_n$,
`(domain` D `(->` D_1 D_2`))`	atomic function domain	$D = D_1 \twoheadrightarrow D_2$,
`(domain` D' `(synonym` D`))`	equivalent domain except in name	

The use of the `synonym` domain definition is the same as for sets.

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. Recall that an atomic domain (Definition 11) is one such that every non-bottom element x is the join of its representation set $Rep(x)$.

Atomic Function Domain

An atomic function domain `(-> E F)` is a set of functions f from `E` to `F` such that:

- f is strict: $f(\bot) = \bot$.
- f is atomic: every image $f(x)$ for $x \in$ `A` is the join of the images $f(x_i)$ over x_is that represent x (i.e., $Rep(x) = \{x_i\}$).

Note that `E` must be an atomic lattice: every x in `E` is $\sqcup Rep(x)$. Note that because every domain that can be defined in Z1 is atomic, any domain already defined in Z1 can be used for `E`.

The domain of functions we have defined has two important properties:

- By definition, every element of an atomic function domain `(-> E F)` can be represented as a function from $Atoms($`E`$)$ to `F`.
- (Property 8) The atomic function domain `(-> E F)` is isomorphic to the product domain `F`^{n} where n is the number of atoms $|Atoms($`E`$)|$ of `E`.

[LIMITATION] Our current implementation of Z1 supports only the case of `(-> E F)` in which `E` is a flat domain `(flat S)`. However, the user can easily model the case in which the left-hand-side is a powerset domain `(2^ S)`. This is because any subset of atoms (a representation set $Rep(\cdot)$) of `(2^ S)` can be expressed from the atom set of `(flat S)`, and because of Property 8. Suppose the user defines `(-> E F)` where `E` is `(flat S)`. Then he can represent an element `f'` of `(-> E' F)`, where `E'` is a powerset domain `(2^ S)` by an element `f` of `(-> E F)` as follows:

- Bottom of `(-> E' F)` is represented by bottom of `(-> E F)`.

- The value of `f'` at $x \in$ `E'` is given by

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

- Updating `f'` value at $x \in$ `E'` by $y \in$ `F` can be expressed, among several possibilities, as

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

depending on the semantics of the operations being modeled.

[NOTE] Suppose that we need a product domain `(domain D (* X ... X))` for which the number of `X`'s is determined by an input program. This domain can be defined as an atomic function domain, where the left hand side of the function is a flat domain of the index set whose upper bound $\rho()$ is determined by a target program:

```
(domain D (-> T X))
(domain T (flat S))
(set S (index ρ)).
```

The Partial Order and Join Operation

- A flat domain `(domain D (flat S))` is the flat lattice $\{\bot, \top\} \cup S$. All $x \in S$ are incomparable. $x \sqsubseteq y$ implies that $x = \bot$, $y = \top$, or $x = y$.
- For a powerset domain, the partial order is set inclusion and the join operation is set union.
- For a product domain `(domain D (* A B))`, the partial order and join operation are defined component-wise: $\langle a, b\rangle \sqsubseteq \langle a', b'\rangle$ iff $a \sqsubseteq a' \;\wedge\; b \sqsubseteq b'$, and similarly for $\sqcup$.
- For an atomic function domain `(domain D (-> A B))`, the partial order and join operation are defined point-wise: $f \sqsubseteq f'$ iff $f(x) \sqsubseteq f'(x)$ for all $x \in A$, and similarly for $\sqcup$.

4.3.3 Element Height and Depth

Domain element height and depth are used in projection expressions. We define the height of an element x to be the length of a maximal chain from bottom to x. The depth of x is defined as the length of a maximal chain from x to top. A chain between two elements is maximal when it cannot be extended to include any more elements. For these definitions to be meaningful, it must be that every maximal chain between two elements have the same length. Recall that this was proved in Property 12 of the domains that can be defined in Z1, in Chapter 3. Recall also Property 13 of our domains: for each domain in Z1, an element's height (respectively depth) can be defined inductively on the domain structure.

This property is important, because using it we can easily compute the height and depth of a domain element for use in projections. 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 $h(x)$. Followings are shown in the proof of Property 13.

- The height of bottom $h(\perp)$ is zero.
- Every atom's height is 1.
- For a flat domain, $h(\top)$ is 2.
- For $x \in$ `(domain D (2^ S))`, $h(x) = |x|$.
- For $x = \langle x_1, x_2 \rangle \in$ `(domain D (* A B))`, $h(x) = h(x_1) + h(x_2)$.
- For an atomic function $f \in$ `(domain D (-> A B))`, $h(f) = h(f(x_1)) + \cdots + h(f(x_n))$ where $\{x_i\} = Atoms(\mathtt{A})$, since $\mathtt{A} \twoheadrightarrow \mathtt{B}$ is isomorphic to $\mathtt{B}^\alpha$ with $\alpha = |Atoms(\mathtt{A})|$ (Property 8).

4.4 Subtypes

Z1 types have a simple subtype relation. Type cast expression in Z1 is permitted only between two types, one of which is a subtype of the other. (Recall that Z1 does not support implicit type coercion; the user must cast types explicitly when necessary.)

- Between domains, there is no subtype relation. Therefore cast from a domain into another domain is always illegal.
- Between sets, the subtype relation is the same as the subset relation.
- Between a domain and a set, a set S is a subtype of its flat domain `(domain D (flat S))`.

[NOTE] `index` and `range` sets are integer subsets, and are thus subtypes of `number`. An `index` or `range` set element can always be cast into `number`. An integer value (`number`) can be cast into a `range` or an `index` set only when the integer is a member of the set.

[NOTE] When `(set` S `(+` $S_1 \cdots S_n$`))`, each S_i is a subtype of S since S_i is a subset of S. An element of S is an element of S_i for some i thus can be cast into S_i.

4.5 Projections

Z1 provides a simple mechanism to tune an analysis: projections of domains. Since the domains are the (abstract) data flow value spaces over which the analysis will be conducted, the smaller the domains the less costly (but less precise) the analysis will be. In Section 2.3 we overviewed the intuition behind the projections and its definition.

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. Z1 arranges that the semantics operate over the projected domain space.

Z1 provides a limited set of projections: all projections project to the top element. A projection is expressed as

`(project` D *pcond*`)`

All elements of D that satisfies the condition *pcond* are projected to $\top_D$. There can be five kinds of projection conditions (*pcond*):

`top?`	$x = \top$
`(height>` n`)`	$\mathrm{height}(x) > n$
`(depth<` n`)`	$\mathrm{depth}(x) < n$ (dual of height)
`(*-and` $pcond_1 \cdots pcond_n$`)`	$x \in D_1 \times \cdots \times D_n.\ \forall i \in \{1, \ldots, n\} : pcond_i(x \downarrow i)$
`(*-or` $pcond_1 \cdots pcond_n$`)`	$x \in D_1 \times \cdots \times D_n.\ \exists i \in \{1, \ldots, n\} : pcond_i(x \downarrow i)$
`(->all` *pcond*`)`	$x \in D_1 \to D_2.\ \forall \mathrm{atom}\ a \in D_1 : pcond(x(a))$
`(->exists` *pcond*`)`	$x \in D_1 \to D_2.\ \exists \mathrm{atom}\ a \in D_1 : pcond(x(a))$

- `(project D (height>` n`))`: project domain elements whose heights are larger than n.

  [Example] Let `D` be a powerset domain `(domain D (2^ S))`. Then, `(project D (height> 1))` simplifies the powerset domain into one equivalent to the flat domain of `S`, since non-singleton elements (whose heights are ¿ 1) are projected into top.

- `(project D (depth<` n`))`: project domain elements whose heights are smaller than n.

- `(project D (*-and` $pcond_1 \cdots pcond_n$`))`: project product domain elements whose ith component satisfy $pcond_i$ for every i.

- `(project D (*-or` $pcond_1 \cdots pcond_n$`))`: project product domain elements whose ith component satisfy $pcond_i$ for some i.

  [Example] Let `D` be a product domain `(domain D (* A B))`. Then, `(project D (*-or top?))` projects an element $\langle$`a`, `b`$\rangle$ of `D` into top if `a` or `b` is top.

- `(project D (->all` *pcond*`))`: project function domain elements whose value at every atom satisfies *pcond*.
- `(project D (->exists` *pcond*`))`: project function domain elements whose value at some atom satisfies *pcond*.

  [Example] Let `(domain D (-> A B))`. Then, `(project D (->exists (height>` n`)))` projects an element f of `D` if the height of $f(x)$ for some atom x is larger than n.

4.6 Functions

After types (sets and domains) are defined, the interpretation definition follows. It is a collection 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.

4.6.1 Function Declarations

Every named function must be declared before its definition. Function `f`'s declaration is:

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

If several functions `f`, `g`, and `h` have the same type, they can be declared in a single declaration:

```
(f g h (from A B C) (to D))
```

`lambda` functions are unnamed functions. Type of a `lambda` function is included in its definition:

$$\boxed{\texttt{(lambda (from } \tau_1 \cdots \tau_n\texttt{) (to } \tau\texttt{) (}x_1 \cdots x_n\texttt{) } e\texttt{)}}$$

The type of every function expression (`function, tfunction, mfunction, lambda`) is from a number of the user-defined/pre-defined types to single such type.

[NOTE] When `A, B, C,` and `D` are the user-defined types,

```
(f (from A B number syntree) (to C D))
```

is an error, since the output type (`(to C D)`) of `f` has more than one user-defined type.

4.6.2 Auxiliary Functions

Auxiliary function can be defined in three categories:

(function f $(x_1 \cdots x_n)$ e)	normal function
(mfunction f $(x_1 \cdots x_n)$ e)	memoized function
(tfunction f (from $S_1 \cdots S_n$) (to τ) $\{(e_1 \cdots e_n\ e)\}^+$)	tabularized function

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

  [NOTE] Z1 requires that the argument types of a tabularized function be sets whose size can be determined independent of the target programs.

  [NOTE] Note that a `tfunction` type is included in its definition, and that its definition does not need formal parameters.

4.6.3 Semantics Function

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

(semantics f $(x_1 \cdots x_n)$ e)

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

(start $e_2 \cdots e_n$)

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`. (Note that the interpretation function is from a program point (`syntree`) and a pre state to a post state.) 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.

4.6.4 Monotonicity

Every function `f` that has a domain (finite-height lattice) value as its argument should be monotonic:

$$\text{for each domain argument type } \tau \text{ and } x, x' \in \tau,$$
$$x \sqsubseteq x' \quad \Longrightarrow \quad \mathtt{f}(\cdots, x, \cdots) \sqsubseteq \mathtt{f}(\cdots, x', \cdots).$$

This monotonicity requirement, together with the fact that every domain is a finite-height lattice, guarantees the termination of the generated analysis.

4.7 Expressions

Every expression is evaluated into 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.

Abstract syntax of Z1 for semantics expression is shown in Figure 4.1 and Figure 4.2.

Certainly, the above expression constructs are not defined for all types of values. Their semantics and requirements of those constructs are as follows.

- Identifiers consist of set names, domain names, set element names, function names, bound names. An identifier is a string of alphabets `a-zA-Z`, numerics `0-9`, and underscore bar `_` . An identifier should start with an alphabet. Lower/upper case letters are differentiated. Z1 uses "`_` " as a prefix for all the generated C identifiers. This prevents an inadvertent naming conflict between the names in a specification and generated C identifiers.

  All identifiers must be different modulo the scope rule of bound names.

- $\boxed{\iota}$

  : an enumerated set element. Its type is its owner set name.

- $\boxed{z}$

  : an integer. It is of type `number`.

- $\boxed{(\tau\ \mathtt{bottom})\text{ and }(\tau\ \mathtt{top})}$

  : these are the bottom $\perp_\tau$ and the top $\top_\tau$ of a domain τ.

- $\boxed{(\tau\ e)}$

  : this is to cast the type of e into τ. Type casting is possible only between the types in subtype relation. Cast into supertype is always legal. Cast into subtype is legal only when the element to cast is included in the subtype.

τ	::=	D | S | `number` | `syntree`	types
f	::=	`(lambda (from` $\tau_1 \cdots \tau_n$`) (to` τ`) (`$x_1 \cdots x_n$`) e)`	
	|	*defined function name*	
e	::= |	ι	enumerated set element
	|	x	bound name
	|	`(_` x ze`)`	indexed bound name x_{ze}
	|	`(`τ `bottom)`	$\bot_\tau$
	|	`(`τ `top)`	$\top_\tau$
	|	`(`τ e`)`	type cast
	|	`(`τ $e_1 \cdots e_n$`)`	powerset elmt $\{e_1, \cdots, e_n\}_\tau$ or product elmt $\langle e_1, \cdots, e_n \rangle_\tau$
	|	`(`τ `(each` x e`))`	ftn lattice elmt $[\lambda x.e]_\tau$
	|	`(!` e $n_1 \cdots n_k$`)`	$e \downarrow n_1 \cdots \downarrow n_k$
	|	`(!` e $\tau_1 \cdots \tau_n$`)`	$e \downarrow \tau_1 \cdots \downarrow \tau_n$
	|	`(@` e $e_1 \cdots e_n$`)`	$e(e_1) \cdots (e_n)$
	|	`(<=` e_1 e_2`)`	$e_1 \sqsubseteq e_2$
	|	`(=` e_1 e_2`)`	equivalence
	|	`(||` e`)`	size $\|e\|$ of a powerset elmt e
	|	`(in` e_1 e_2`)`	$e_1 \in e_2$
	|	`(join` $e_1 \ldots e_n$`)`	$e_1 \sqcup \cdots \sqcup e_n$
	|	`(join (for` x ze_1 ze_2 e`))`	$\sqcup_{ze_1 \leq x \leq ze_2} e$
	|	`(mapjoin` f e`)`	$\sqcup_{x \in e} f(x)$
	|	`([/]` e `(by` e_{1_1} e_{1_2}`)`$\cdots$`(by` e_{n_1} e_{n_2}`))`	$e[e_{1_2}/e_{1_1}] \cdots [e_{n_2}/e_{n_1}]$
	|	`(map[/]` e f e_1`)`	$e[f(x)/x]_{x \in e_1}$
	|	`(map[/]` e f g e_1`)`	$e[f(x)/g(x)]_{x \in e_1}$
	|	`(dom` e`)`	$\{\, x \mid e(x) \neq \bot \,\}_\tau$
	|	`(nth` e ze`)`	ze-th elmt of a powerset elmt e
	|	`(`f $e_1 \cdots e_n$`)`	function apply $f(e_1, \cdots, e_n)$
	|	`(if` ze e_1 e_2`)`	conditional
	|	`(switch` $\{ze\ e\}^+$ `default` e`)`	switch to the first true branch
	|	`(typecase` e $\{\tau\ e'\}^+$`)`	switch to matched type branch
	|	`(begin` be^+ `e)`	sequential bindings and body
	|	`error`	semantics error
	|	ze	integer valued expression
be	::=	`(let` x tag e`)`	binding $x = e$
	|	`(let (_` x i`)` tag `(for` i ze_1 ze_2 e`))`	indexed binding $x_i = e$ for $ze_1 \leq i \leq ze_2$
	|	`(let (_` x z`)` tag e_1 `(_` x i`)` tag `(for` i ze_1 ze_2 e_2`))`	indexed binding with initialization $x_z = e_1$ with $x_i = e_2$ for $ze_1 \leq i \leq ze_2$

Figure 4.1: Abstract Syntax of Z1 for Semantics Expression (part 1)

$$
\begin{array}{llll}
tag & ::= & :\tau & \\
 & | & empty & \\
ze & ::= & x & \text{bound name} \\
 & | & z & \text{integer constant} \\
 & | & \texttt{(and } ze_1\ ze_2\texttt{)} \mid \texttt{(or } ze_1\ ze_2\texttt{)} \mid \texttt{(not } ze\texttt{)} & \\
 & | & \texttt{(}rop\ ze_1\ ze_2\texttt{)} \mid \texttt{(}bop\ ze_1\ ze_2\texttt{)} & \\
 & | & \texttt{(}bop\ \texttt{(for } i\ ze_1\ ze_2\ ze_3\texttt{))} & bop_{i=ze_1}^{ze_2} ze_3 \\
bop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} & \text{integer binary operation} \\
rop & ::= & \texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} & \text{integer relation}
\end{array}
$$

Figure 4.2: Abstract Syntax of Z1 for Semantics Expression (part 2)

[Example] Suppose S is an integer range set and z is a number. `(`S z`)` casts an integer z into an element of the range set S. In this case z should be in the range of S.

[Example] Suppose `(domain` D `(flat` S`))`. Cast from S to D is always legal, while cast from D to S is legal only when the element to cast is neither $\bot_D$ nor $\top_D$.

[Example] Suppose `(set` S `(+` S_1 S_2`))` and $x \in S_1$. Then, `(`S x`)` casts x into S. In this case, this cast expression is equivalent to the injection operation. Suppose $y \in S$ and suppose its injected value is of type S_1. Then, `(`S_1 y`)` casts the injected value of S into S_1.

- $\boxed{(\tau\ e_1 \cdots e_n)}$

  : depending on τ, this evaluates to an element $\{e_1, \cdots, e_n\}_\tau$ of a powerset domain or an element $\langle e_1, \cdots, e_n\rangle_\tau$ of a product domain or a product set. In case of a powerset domain `(domain` τ `(2^` S`))`, all e_i's should be of type S. In case of a product domain/set `(domain` τ `(*` $\tau_1 \cdots \tau_n$`))`, each e_i's type should be τ_i.

  [Example] Let `(set` S `(elements apple orange mellon))` and let `(domain` D `(2^` S`))`. Then, `(`D `apple orange)` is an element $\{\texttt{apple}, \texttt{orange}\}$ of D.

  [Example] Let $e_1 \in D_1$ and $e_2 \in D_2$ and let `(domain` D `(*` D_1 D_2`))`. Then, `(`D e_1 e_2`)` is an element $\langle e_1, e_2\rangle \in D$.

- $\boxed{(\tau\ \texttt{(each}\ x\ e\texttt{))}}$

  : an element of an atomic function domain `(domain` τ `(->` D_1 D_2`))`. The x ranges over the set of atoms of D_1. The type of e, with x being of type D_1, should be D_2.

  [Example] Let `(domain D (-> A B))`, f be an element of `D`, and b of `B`. Then, `(D (each x (join (@ f x) b)))` is a function f' of `D` where $f'(x) = f(x) \sqcup b$ for each $x \in$ `A`.

- $\boxed{\texttt{(! } e\ n_1 \cdots n_k\texttt{) or (! } e\ \tau_1 \cdots \tau_k\texttt{)}}$

  : component selection (left associated) $e \downarrow n_1 \cdots \downarrow n_k$. The type of e should be either a product domain or a product set. The selection is done either by a number from 1 or by a component type name.

  [Example] Let e be an element of `(domain D (* A B))`. `(!` e `A)` is e's `A` component. When `(domain A (* V W))`, `(!` e `A V)` is equivalent to `(! (!` e `A) V)`. `(!` e `1)` and `(!` e `1 2)` have the same effect.

- $\boxed{\texttt{(@ } e\ e_1 \cdots e_n\texttt{)}}$

  : probing an atomic function element e's value at e_i's: $e(e_1)\cdots(e_n)$. The type of e should be $\tau_1 \rightarrow (\tau_2 \rightarrow \cdots \tau_{n+1})$. The type of e_i should be τ_i. The resulting value type is τ_{n+1}.

  [Example] Let `(domain D (-> E G))`, f be an element of `D`, and e of `E`. `(@` f e`)` is f's value ($\in$`G`) at e. Let `(domain G (-> V W))` and v be an element of `V`. `(@` f e v`)` is equivalent to `(@ (@` f e`)` v`)`.

- $\boxed{\texttt{(<= } e_1\ e_2\texttt{) or (= } e_1\ e_2\texttt{)}}$

  : partial order (or equivalence) predicate. Types of e_i's should be the same domain type. Result value is either 0 or 1, whose type is `number`.

- $\boxed{\texttt{(|| } e\texttt{)}}$

  : size of an powerset domain element. Type of e should be `(domain D (2^ S))`. Result type is `number`.

- $\boxed{\texttt{(in } e_1\ e_2\texttt{)}}$

  : predicate to see if a set S's element e_1 is included in $e_2 \in$ `(domain D (2^ S))`. Result type is `number` (0 or 1).

- $\boxed{\texttt{(join } e_1 \ldots e_n\texttt{)}}$

  : join operation for domain elements. All e_i's should be of the same domain type.

- $\boxed{\texttt{(join (for } x\ ze_1\ ze_2\ e\texttt{))}}$

  : join operation for domain elements. The type τ of e, with x's type being `number`, should be a domain. When $ze_1 < ze_2$ result is the bottom $\perp_\tau$ of τ.

  [Example] Suppose f is a function expression which returns an element of a domain `D` from an element of `number`. `(join (for i 1 3 (`f `i)))` is $f(1) \sqcup f(2) \sqcup f(3)$.

  [Example] Suppose we need to collect parameters ($\in$ `(domain Ids (2^ Id))`) of a procedure `p` in a target program. Suppose `NthParam(n,p)` is the `n`-th parameter's identifier index ($\in$ `Id`), and suppose `NoParams(p)` is the number of parameters of `p`. Then

  ```
  (join (for i 1 (NoParams p) (Ids (NthParam i p))))
  ```

  is an element of `Ids` which consists of the parameter indices of `p`.

- $\boxed{\texttt{(mapjoin } f\ e\texttt{)}}$

  : join operation after applying f on each element of e: `(join (`f e_1`)...(`f e_n`))` where $\{e_i\} = e$. Type of e should be a powerset domain `(domain D (2^ S))`. Map function f's type should be `(from` S`) (to` D`)`. When $e = \bot_\tau$, result is $\bot_\tau$.

- $\boxed{\texttt{([/] } e\ \texttt{(by } e_{1_1}\ e_{1_2}\texttt{)}\cdots\texttt{(by } e_{n_1}\ e_{n_2}\texttt{))}}$

  : an atomic function domain element e's entry on e_{i_1} is changed to e_{i_2} (left associative):

  $$e[e_{1_2}/e_{1_1}]\cdots[e_{n_2}/e_{n_1}].$$

  The type of e should be `(D (-> E F))`. All e_{i_1} should be atoms of `E`. Types of e_{i_2} should be `F`.

- $\boxed{\texttt{(map[/] } e\ f\ e_1\texttt{)}}$

  : update an atomic function domain element e at x_i by $f(x_i)$ where $e_1 = \{x_i\}$:

  $$e[f(x_1)/x_1]\cdots[f(x_n)/x_n] \quad \text{where} \quad e_1 = \{x_i\}.$$

  The type of e should be `(domain D (-> E F))` with `(E (flat S))`. Type of e_1 should be a powerset domain `(2^ S)`. Type of f should be `(from S) (to F)`. e's entry at `(E` x`)` for each $x \in e_1$ is changed to $f(x)$. When $e_1 = \bot$, result is e.

  `(map[/]` e f e_1`)` is equivalent to

  `([/]` e `(by (E` x_1`) (`f x_1`))` ... `(by (E` x_n`) (`f x_n`)))` where $e_1 = \{x_i\}$.

- $\boxed{\texttt{(map[/] } e\ f\ g\ e_1\texttt{)}}$

  : update an atomic function domain element e at $g(x_i)$ by $f(x_i)$ where $e_1 = \{x_i\}$:

  $$e[f(x_1)/g(x_1)]\cdots[f(x_n)/g(x_n)] \quad \text{where} \quad e_1 = \{x_i\}.$$

  The type of e should be `(domain D (-> E F))` with `(E (flat S))`. Type of e_1 should be a powerset domain `(2^ S)`. Type of f should be `(from S) (to F)`. Type of g should be `(from S) (to E)`. When $e_1 = \bot$, result is e.

  `(map[/]` e f g e_1`)` is equivalent to

  `([/]` e `(by (`g x_1`) (`f x_1`))` ... `(by (`g e_n`) (`f e_n`)))` where $e_1 = \{x_i\}$.

- $\boxed{\texttt{(dom } e\texttt{)}}$

  : collection of atoms on which an atomic function domain element e is defined. The type of e should be `(domain D (-> E F))` with `(domain E (flat S))`. Result type is one defined as `(2^ S)`. (There should be a unique such type `(domain E' (2^ S))`.) When $e = \bot_\tau$, result is $\bot_{\tau'}$.

  [Example] Suppose $f \in$ `(domain D (-> E F))` has non-bottom values for atoms e_1 and e_2. `(dom` f`)` is `(join` e_1 e_2`)` when `E` is a powerset domain. If `(domain E (flat S))` and `(domain E' (2^ S))`, then `(dom` f`)` is `(join (E' (S` e_1`)) (E' (S` e_2`)))`.

- $\boxed{\texttt{(nth } e\ ze\texttt{)}}$

  : choose ze-th element of e. The type of e should be `(domain D (2^ S))`. Result type is S. It is guaranteed that all constituents of e can be selected by choosing from 1 to `(||` e`)`.

- $\boxed{(f\ e_1 \cdots e_n)}$

  : function application. Type of f should be `(from` $\tau_1 \cdots \tau_n$`) (to` τ`)` and that of e_i should be τ_i. The result is of type τ.

- $\boxed{\texttt{(if } ze\ e_1\ e_2\texttt{)}}$

  : depending on the value (`number`) of ze, evaluate e_1 (true branch) or e_2 (false branch). The types of e_1 and e_2 should be the same, which is the result type.

- $\boxed{\texttt{(switch } \{ze\ e\}^+ \texttt{ default } e\texttt{)}}$

  : switches to the first true branch. Evaluate sequentially each condition ze, if it is non-zero, evaluates its branch e. The types of the branch bodies should all be the same, which is the result type.

- $\boxed{\texttt{(typecase } e\ \tau_1\ e_1 \ldots \tau_n\ e_n\texttt{)}}$

  : switches to the matched type branch. Type of e should be a union set `(set` τ `(+` $\tau_1 \cdots \tau_m$`))`. Depending on the injected value's type of e, it evaluates corresponding e_i. All e_i's types should be the same. If no matched type, the analysis will abort.

- $\boxed{\texttt{(begin } be^+ \texttt{ e)}}$

  : sequential bindings and an expression of their scope. Sequentially bind each name to its value in be and evaluates the body e. After the evaluation of e the bindings are freed.

- $\boxed{\texttt{error}}$

  : semantics error. Its type is "any type": `error` can be anywhere without causing any type conflict. Z1 generates C code to abort the analysis.

- $\boxed{\texttt{(let } x\ e\texttt{)}}$

  : new binding for x. x's value is that of (and the type) of e.

- $\boxed{\texttt{(let (\_ } x\ i\texttt{) (for } i\ ze_1\ ze_2\ e\texttt{))}}$

  : new binding for indexed names x_i's. x_i's value (and the type) is that of e, where i ranges from ze_1 to ze_2 (inclusive). ze_i's should be non-negative integers and should be $ze_1 \leq ze_2$. Z1 generates code to check these requirements at run-time. The indexing variable i and the indexed variable x should not be free inside the index bound expressions ze_1 and ze_2. The indexed variable x should not be free in binding-value expression e.

[Example] This binding expression is used for, e.g.,

$$x_i = y_i \downarrow 1, \text{ for } i = 1, \ldots, 10.$$

In Z1, the above is `(let (_ x i) (for i 1 10 (! (_ y i) 1)))`

[Example] Expression `(let (_ x i) (for i -3 3 ...))` causes an error since index range contains negative value.

[Example] The following expression causes an error.

```
(let (_ x i) (for  i (+ i 1) (+ x 10)
                   (+ (_ y i) x)))
```

Index's upper/lower bound expressions `(+ i 1)` and `(+ x 10)` contains the indexing variable `i` and the indexed variable `x` as free variables, which is prohibited. Another reason is that the binding-value expression `(+ (_ y i) x)` has `x` as a free variable, which is also prohibited.

- `(let (_` x z`)` e_1 `(_` x i`) (for` i ze_1 ze_2 e_2`))`

: bind indexed names x_i's to the value (and the type) of e_2, with initial binding $x_z = e_1$, where i ranges from ze_1 to ze_2 (inclusive). ze_i's should be non-negative integers and should be $ze_1 \leq ze_2$. Z1 generates code (optional) to check these requirements at run-time. The types of e_1, e_2, and tags (optional) should all be the same. The indexing variable i and the indexed variable x should not be free in the index bound expressions ze_1, ze_2, and initial binding-value expression e_1.

[Example] This binding expression is used for, e.g.,

$$\begin{cases} x_0 = c; \\ x_i = \ldots x_{i-1} \ldots, & \text{for } i = 1, \ldots, 10. \end{cases}$$

In Z1, the above is

```
(let  (_ x 0) c
      (_ x i) (for i 1 10 (...(_ x (- i 1))...)))
```

[Example] Following expression causes an error.

```
(let   (_ x -1) ...
       (_ x i) (for i -1 10 ...)
```

It is because the index has negative value.

[Example] Following expression causes an error.

```
(let   (_ x 0) (+ x i)
       (_ x i) (for i (+ i 1) (+ x 10) e))
```

The expression to be bound with `(_ x 0)` has the indexed variable `x` and the indexing variable `i` as free variables, which is prohibited. Another reason is that the index variable's lower/upper bound expressions `(+ i 1)` and `(+ x 10)` have `i` and `x` as free variables, which is also prohibited.

4.8 Foreign Procedures

Only C procedures can be imported into Z1. As other Z1 functions, foreign procedures need be declared before their use. Declaring C procedures in Z1 is in the same way as for other Z1 functions.

Because no foreign types are allowed inside Z1, two predefined type names, `syntree` and `number`, may be used to declare the types of those procedures.

The correspondence between C types and these two predefined Z1 types must be defined in a C header file

```
user.h
```

[Example] Suppose the user has a C procedure `isPlus`, which returns true when it's argument (a syntax tree node) is a plus operation node. Inside Z1, 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;
typedef int number;
```

4.9 The Abstract Syntax of a Target Language

Z1 does not provide any construct to specify the abstract syntax of a target language. System Z1 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 use them as foreign procedures.

Currently, we have implemented ANSI C, FORTRAN, and SCHEME front-ends for an intermediate language MIL (MIPRAC [HA92] Intermediate Language), along with a set of routines to access MIL programs, so that Z1 may be used to analyze programs in these source languages.

4.10 Syntax of Z1

4.10.1 Lexical Rule

Lexical rule for identifier, number, and comment is the following.

initial	[a-zA-Z]
subsequent	({initial}|[0-9]|"_ ")
id	{initial}({subsequent})*
int	[-+]?[0-9]+
comment	;[^ newline]*

Reserved strings are: (separated by a blank)

```
types domain set project synonym flat range index elements number syntree
start semantics function mfunction tfunction bottom top each nth by
dom join mapjoin map[/] begin if switch typecase default error let
for lambda from to in not or and - / = < > <= >= * -> + 2^ top?  height>
depth< *-and *-or ->all ->exists ( ) :  _ !  @ [/] ||
```

4.10.2 Parsing Rule

Uppercase names are non-terminal symbols.

```
SPEC     :=   TYPE_SPEC SEM_SPEC
         |    TYPE_SPEC
TYPE_SPEC :=    ( types T_SPECS )
T_SPECS   :=    T_SPEC
          |     T_SPECS T_SPEC
T_SPEC    :=    DOMAIN_SPEC
          |     SET_SPEC
          |     PROJECT_SPEC
DOMAIN_SPEC :=  ( domain ID DOMAIN_EXPR )
DOMAIN_EXPR :=  ( synonym ID )
            |   ( -> ID ID )
            |   ( * IDS )
            |   ( 2^ ID )
            |   ( flat ID )
ID  :=       TID
IDS :=       ID
    |        IDS ID
SET_SPEC   :=       ( TSET ID SET_EXPR )
SET_EXPR   :=       ( range NUMBER NUMBER )
           |        ( index ID )
           |        ( elements IDS )
           |        ( * IDS )
```

```
           |        ( + IDS )
           |        ( synonym ID )
NUMBERS :=       NUMBER
        |        NUMBERS NUMBER
PROJECT_SPEC :=       ( project ID PROJECT_EXPR )
PROJECT_EXPR :=       top?
            |        ( height> NUMBER )
            |        ( depth< NUMBER )
            |        ( *-and PROJECT_EXPRS )
            |        ( *-or PROJECT_EXPRS )
            |        ( ->all PROJECT_EXPR )
            |        ( ->exists PROJECT_EXPR )
PROJECT_EXPRS :=       PROJECT_EXPR
             |        PROJECT_EXPRS PROJECT_EXPR
SEM_SPEC :=       S_SPECS
S_SPECS  :=       S_SPEC
         |        S_SPECS S_SPEC
S_SPEC   :=       DECL
         |        FTN_SPEC
DECL     :=        ( IDS FROM TO )
         |        ( start EXPRS )
FTN_SPEC :=       ( semantics ID PARAM EXPR )
         |        ( function ID PARAM EXPR )
         |        ( mfunction ID PARAM EXPR )
         |        ( tfunction ID FROM TO TUPLES )
TUPLES  :=        ( EXPRS )
        |        TUPLES ( EXPRS )
FROM    :=        ( from TYPES )
        |        ( from )
TO      :=       ( to TYPES )
PARAM   :=       ( IDS )
        |        ( )
TYPES   :=       TYPE
        |        TYPES TYPE
TYPE    :=       ID
        |        number
        |        syntree
TYPE_TAG  :=
          |        : TYPE
EXPRS     :=       EXPR
          |        EXPRS EXPR
EXPR      :=       ID
          |        ( _ ID EXPR )
          |        ( ID bottom )
          |        ( TDOM EXPR )
          |        ( ID top )
          |        ( ID ( each ID EXPR ) )
          |        ( ID EXPR )
          |        ( number EXPR )
          |        ( ID EXPR EXPRS )
          |        ( ID )
          |        ( LAM_EXPR EXPRS )
```

```
          |        ( ! EXPR SELECTORS )
          |        ( @ EXPR EXPRS )
          |        ( nth EXPR EXPR )
          |        ( || EXPR )
          |        ( [/] EXPR BY_EXPRS )
          |        ( join EXPR EXPRS )
          |        ( mapjoin ID EXPR )
          |        ( mapjoin LAM_EXPR EXPR )
          |        ( map[/] EXPR FTN_EXPR EXPR )
          |        ( map[/] EXPR FTN_EXPR FTN_EXPR EXPR )
          |        ( begin BIND_EXPRS EXPR )
          |        ( if EXPR EXPR EXPR )
          |        ( switch SWITCH_EXPRS default EXPR )
          |        ( typecase EXPR TYPECASE_EXPRS )
          |        error
          |        ( <=  EXPR EXPR )
          |        ( <  EXPR EXPR )
          |        ( >=  EXPR EXPR )
          |        ( >  EXPR EXPR )
          |        ( = EXPR EXPR )
          |        ( in EXPR EXPR )
          |        ( not EXPR )
          |        ( or EXPR EXPRS )
          |        ( and EXPR EXPRS )
          |        ( + EXPR EXPR )
          |        ( - EXPR EXPR )
          |        ( * EXPR EXPR )
          |        ( / EXPR EXPR )
          |        ( join ( for ID EXPR EXPR EXPR ) )
          |        ( + ( for ID EXPR EXPR EXPR ) )
          |        NUMBER
SELECTORS  :=       ID
           |        NUMBER
           |        SELECTORS ID
           |        SELECTORS NUMBER
BIND_EXPRS :=       BIND_EXPR
           |        BIND_EXPRS BIND_EXPR
BIND_EXPR  :=       ( let ID TYPE_TAG EXPR )
           |        ( let ( _ ID ID ) TYPE_TAG ( for ID EXPR EXPR EXPR ) )
           |        ( let ( _ ID NUMBER ) TYPE_TAG EXPR
                          ( _ ID ID ) TYPE_TAG ( for ID EXPR EXPR EXPR ) )
SWITCH_EXPRS:=       SWITCH_EXPR
            |        SWITCH_EXPRS SWITCH_EXPR
SWITCH_EXPR :=       EXPR EXPR
TYPECASE_EXPRS     :=  TYPECASE_EXPR
                   |   TYPECASE_EXPRS TYPECASE_EXPR
TYPECASE_EXPR      := ID EXPR
BY_EXPRS   :=   BY_EXPR
           |    BY_EXPRS BY_EXPR
BY_EXPR    :=       ( by EXPR EXPR )
FTN_EXPR   :=   ID
           |    LAM_EXPR
```

```
LAM_EXPR   :=   ( lambda FROM TO PARAM EXPR )
```

4.11 A Specification Example

For expository purpose, let us consider the following toy language. Expressions have integer values, statements change the machine state, and a program is a sequence of statements.

$$
\begin{array}{lcll}
\mathit{Expr} & ::= & x & \text{variable} \\
 & | & c & \text{constant} \\
 & | & \mathit{Expr}\ \texttt{+}\ \mathit{Expr} & \text{addition} \\
\mathit{Stmt} & ::= & x\ \texttt{=}\ \mathit{Expr}; & \text{assignment} \\
 & | & \texttt{if}\ \mathit{Expr}\ \mathit{Stmt_1}\ \mathit{Stmt_2}; & \text{if} \\
 & | & \texttt{while}\ \mathit{Expr}\ \mathit{Stmt}; & \text{while loop} \\
\mathit{Pgm} & ::= & \mathit{Stmt}^{+} &
\end{array}
$$

Suppose we define an abstract interpreter of the toy language for the constant propagation analysis. The semantics function type will be

$$\texttt{syntree} \times \mathit{store} \rightarrow \mathit{value} \times \mathit{store}$$

For each program expression (`syntree`) and store state, the interpretation function returns the expression value and next store state.

Firstly, the domain definitions:

```
; Domain and set definitions
;
(domain POST (* V S))     ; output type of the semantics function Eval
                          ; product of value and store.
(domain S (-> Id V))      ; store: function from identifier to a value
; Value domain is defined as a powerset domain.
; The accompanied projection expression trims the powerset structure into
; flat one which is conventional in constant propagation analysis.
(domain V (2^ Z1))        (project V (height> 1))
(domain Id (flat Var))          ; identifier domain
(set Z1 (range -99999 99999))   ; finite set of integers
(set Var (system NumOfVars))    ; set of variable indices 0,...,NumOfVars()
                                ; of a given input program.
                                ; User should write C procedure NumOfVars().
```

The output type of the semantics function is defined `(domain POST (* V S))` as a product of the value and the store. Abstract store `S` is an atomic function domain from identifiers to their values. Abstract value is a powerset domain `(domain V (2^ Z1))` of a finite range set `Z1`. As in conventional constant propagation analysis, we might have used the flat lattice structure (domain V (flat Z1)). However, by defining an abstract interpreter over the powerset domain, and later, by applying projections to the domain, we will be able to have a variety of analyses which are different in cost and accuracy from the conventional constant propagation analysis. For the conventional constant propagation analysis, we add a projection

(project V (height> 1)).

This projects all elements of V that contain more than 1 integer value into top.

Identifier domain Id is a flat domain from the set Var of variables of an input program. Set Var is an index set for the variables in the input program. The C procedure call NumOfVars() returns the number of variables in the input program.

Next, we need several functions that operates over the abstract syntax tree of the input program. These foreign procedures, which will be defined in C by user, are declared inside Z1:

```
; Foreign procedure declarations.
; These functions should be provided by the user as C procedures.
; C types for syntree and number should also be defined in user.h
(isVar isConst isEqual isPlus isAssign isIf isWhile (from syntree) (to number))
(Xid (from syntree) (to number))
(Xconst (from syntree) (to number))
(NthSubexpr (from syntree number) (to syntree))
(AssignExpr IfCond IfTrue IfFalse WhileCond WhileBody (from syntree) (to syntree))
```

The functions (isVar,isConst, etc.) in the first group are predicates to determine the syntactic category of an abstract syntax node: hence the type is from syntree to number (0 or 1). Xid returns the identifier index for a variable expression node. Xconst returns the number value of a constant expression node. NthSubexpr selects n-th subexpression syntax node of the current node. The last set of functions (AssignExpr, IfCond, etc.) selects appropriate subexpressions from an expression.

Next, we have the following auxiliary functions:

```
; Auxiliary functions
;
; abstraction of an integer to V (abstract integer)
(absToV (from number) (to V))
(function absToV (n)
  (if (or (< n -99999) (> n 99999)) (V top)
    (V (Z1 n))))

; plus operation over V (abstract integer)
(plusV (from V V) (to V))
(function plusV (v1 v2)
  (if (or (= v1 (V bottom)) (= v2 (V bottom))) (V bottom)
    (if (or (= v1 (V top)) (= v2 (V top))) (V top)
      ; Add each pair <z1i,z2j> from v1 = {z11,...,z1n} and v2 = {z21,...,z2m}.
```

```
        ; We must first cast element z into number, apply +, and
        ; recast the result into V.
        (mapjoin (lambda (from Z1) (to V) (z1)
                   (mapjoin (lambda (from Z1) (to V) (z2)
                              (absToV (+ (number z1)
                                         (number z2))))
                            v2))
                 v1))))
```

The first auxiliary function `absToV` is in order to transform a value of `number` into its abstract counterpart of `V`. It checks if the number is within the range of the set `Z1`, if true, the number is cast into `V`: `(V (Z1 n))`, otherwise, we use top: `(V top)`. The `plusV` is an abstract version of the integer addition. Since an abstract value is a set of numbers, the abstract plus operations "adds" every pair from the two abstract operands. Note that each element of an abstract value is of type `Z1`. We cast them into `number` (`(number z)`), apply `+` to those numbers, and cast the result (of type `number`) back into `V` by `absToV`.

Now, the semantics function `Eval` is defined.

```
; Semantics function Eval: (syntree * S) -> (V * S)
; Given each construct and a store, Eval returns the value and new store
(Eval (from syntree S) (to POST))
(semantics Eval (x s)
  (switch
  (isVar x)    (POST (@ s (Id (Var (Xid x))) s))    ; store's entry of x
  (isConst x)  (POST (absToV (Xconst x)) s)  ; constant value
  (isPlus x)   (begin
                (let POST1 (Eval (NthSubexpr x 1) s)) ; eval operand1
                (let v1 (! POST1 V))
                (let POST2 (Eval (NthSubexpr x 2) (! POST S))) ;eval operand2
                (let v2 (! POST2 V))
                ; plus the operands, store is the last one
                (POST (plusV v1 v2) (! POST2 S)))
  (isAssign x)  (begin
                 ; eval the assigning value
                 (let POST1 (Eval (AssignExpr x) s))
                 (let id (AssignVar x))
                 ;overwrite store entry of id with the new value
                 (let s2 ([/] (! POST1 S) (by id (! POST1 V))))
                 (POST (V bottom) s2))
  (isIf x)  (begin
              (let POST1 (Eval (IfCond x) s))
              (let cond (! POST1 V))
              ; if cond value is 1 or top, evaluate the true branch.
              (let POST2 (if (<= (absToV 1) cond)
                              (Eval (IfTrue x) (! POST1 S))
                             (POST bottom)))
              ; if cond value is 0 or top, evaluate the false branch.
              (let POST3 (if (<= (absToV 0) cond)
                              (Eval (IfFalse x) (! POST1 S))
                             (POST bottom)))
```

```
             ; join the two branch results.
             (join POST2 POST3))
  (isWhile x) (begin
              (let POST1 (Eval (WhileCond x) s))
              (let cond (! POST1 V))
              (switch
                 (= cond (top V))
                 (join POST1
                       (Eval x (! (Eval (Whilebody x) (! POST1 S)) S)))
                  (= cond (absToV 1))
                 (Eval x (! (Eval (WhileBody x) (! POST1 S)) S))
                  (= cond (absToV 0)) POST1
                  default (POST bottom)
              ))
  ))
```

For each syntactic category of a given syntax node `x`, `Eval` defines an abstract interpretation rule. In case of a variable `(isVar x)`, store's entry for the variable is the output value. Note that the variable node `x`'s identifier index is extracted by `(Xid x)` which is of type `number`. In order to apply the store, the identifier index should be cast into the type `Id`: `(Id (Var (Xid x)))`.

In case of while expression `(isWhile x)`, we need to evaluate the while condition `(Eval (WhileCond x) s)`. When the condition value is top, we join two possibilities: evaluation result of the while loop, and the result without iterating the while loop. When the condition value is precisely true or false we evaluate corresponding case.

All other cases are similarly straightforward.

Remember that the generated analyzer computes, for each program point, a pre-state and a post-state which characterize possible cases that will occur in execution of an input program:

$$\mathit{Tabulate}(F_{\mathtt{Eval}}, \Sigma_P, \bot_{\mathtt{S}}) \qquad \text{(see Figure 2.1).}$$

The pre-state corresponds to `Eval`'s input type excluding the `syntree` component. The post-state corresponds to its output type. A pre-state (respectively a post-state) is, in our case, an element of `S` (respectively an element of `POST`).

The store components of the post-states that are associated with the input program points will show us which variable stays constant.

Chapter 5

Analysis Specifications and Experiments

It needs to be seen whether the analysis specification language Z1 is powerful enough for various program analyses for a wide range of target languages. In this chapter we will present several analysis specifications (in Z1) and analysis examples which are conducted by the generated analyzers with varying degree of projections. These experiments may substantiate the claim that

- system Z1 can generate detailed analyzers for practical target languages, and that
- projection expressions are useful for managing the accuracy-cost tradeoff of the analysis.

Using Z1, both conventional and non-conventional data flow analyses for the target languages like C have been implemented. We present these analyses below. We also show how projections are used to manage their accuracy and cost. We discuss a case in which the analysis cost is reduced by projections without a loss of accuracy. The target language for all our examples is an intermediate language called MIL (MIPRAC Intermediate Language [HA92]), into which C, FORTRAN, and SCHEME programs are translated. The generated analyzers do not have any restriction regarding the input program text; they perform an interprocedural analysis of any legitimate program in the target language. We have implemented the following analyses:

- **Alias analysis.** During program execution, distinct names or expressions can refer to the same memory location. Alias analysis computes which expressions may evaluate to the same memory location during execution.
- **Constant propagation.** An expression can return a constant value during execution. This information can, in turn, cause other expressions to be constant-valued. Constant

propagation determines which expressions are constant-valued and what their values are.

- **Def-use/use-def chain analysis**. Suppose that one expression writes a value to a memory location, and later another expression accesses the value in the location. Def-use chain analysis computes this dependence relation between the expressions. Use-def chain relation is its dual one, which can be derived off-line by the inversion of the def-use chain information. By def-use/use-def chain analysis, we can determine interprocedural side-effects: e.g., procedure A defines a memory location and procedure B reads or modifies the value at that location.

- **Memory object lifetime analysis**. This is a duplication by Z1 of the one in [YH93]. During program executions some memory is allocated, used, and deallocated. Procedure parameters, local variables, malloc'd objects, and cons cells are of such a kind. This analysis computes the longevity and temporal locality of each allocated memory object: the interval from its creation until the access points. The interval from the creation time until the last access point is the longevity of an object.

5.1 The Target Language MIL

All the analyses in this chapter have MIL (MIPRAC [HA92] Intermediate Language) as the target language. That is, the generated analyzers analyze source programs written in MIL. Since we have ANSI C, FORTRAN, and SCHEME front-ends already built for MIL, our analyses analyze programs written in those languages.

The abstract syntax of MIL is shown in Figure 5.1. A program is an expression. *Const* stands for integer constants and *Id* for identifiers. Identifiers of a program are the distinct names that are assigned to the variables and the `create` expressions in a program. Any composite access to a data structure can be expressed using `read` and `write` expressions. A variable (an expression consisting of a single identifier) evaluates to its location in memory. An arithmetic expression is used to express an offset from a base location. For example,

```
(read [(+ x 2) 4])
```

reads four bytes from the location which is offset by two bytes from the address of `x`. Any size of memory block can be allocated by a `create` expression. There are three types of values: integer, location, and closure. (We do not consider floating point values.) Every expression evaluates to one of these values. Constants in square brackets in `read`, `call`, and declarations specify the number of bytes to read, to return, and to allocate, respectively.

The important feature of the language is that

- It has no iterative construct. All iterative expressions is expressed by tail-recursive procedure calls.

$$
\begin{array}{lcl}
\mathit{Expr} & ::= & (\texttt{begin}\ \mathit{Expr}^{+}) \\
 & | & (\texttt{if}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr}) \\
 & | & (\texttt{read}\ \mathit{Expr}) \\
 & | & (\texttt{read}\ [\mathit{Expr}\ \mathit{Const}]) \\
 & | & (\texttt{write}\ \mathit{Expr}\ \mathit{Expr}) \\
 & | & (\texttt{create}\ \mathit{Expr}\ \mathit{Id}) \\
 & | & (\texttt{call}\ \mathit{Expr}\ \mathit{Expr}^{*}) \\
 & | & (\texttt{call}\ [\mathit{Expr}\ \mathit{Const}]\ \mathit{Expr}^{*}) \\
 & | & (\texttt{op}\ \mathit{Expr}\ \mathit{Expr}) \\
 & | & (\texttt{procedure}\ \mathit{Params}\ \mathit{Locals}\ \mathit{Expr}) \\
 & | & \mathit{Id} \\
 & | & \mathit{Const} \\
 & | & [\mathit{Const}\ \mathit{Const}] \\
\mathit{Params} & ::= & (\texttt{parameter}\ \mathit{Decl}^{+}) \mid \epsilon \\
\mathit{Locals} & ::= & (\texttt{local}\ \mathit{Decl}^{+}) \mid \epsilon \\
\mathit{Decl} & ::= & \mathit{Id} \mid [\mathit{Id}\ \mathit{Const}]
\end{array}
$$

Figure 5.1: Abstract Syntax of MIL

- Only two control constructs are available in MIL: procedure call and conditional branch. It has no `goto` or `return`. These control constructs in the source programs are removed by the MIPRAC front-ends using the control flow normalization method in [Amm92].

The reader may want to see [HA92] or [Har92] for the definition of MIL. An informal semantics for each expression constructs is as follows. We need to understand the standard semantics of MIL because we will define abstract interpreters (approximate semantics) for it later.

- (`begin` $e_0 \cdots e_n$): Evaluate the expressions in sequence. Return the value of the last expression e_n.
- (`if` e_0 e_1 e_2): Evaluate e_0 to see the truth value (an integer value). If true (non zero), evaluate e_1 and return its value, else evaluate e_2 and return its value.
- (`read` e): Evaluate e to get the location to read. Fetch the value from the location in the memory. The number of bytes that we fetch from the memory is the size of an integer, by default. Return the value read.
- (`read` $[e\ \kappa]$): Evaluate e to get the location to read. Fetch the value from the location in the memory. The number of bytes that we fetch from the memory is k. Return the value read.

- (`write` e_0 e_1): Evaluate e_0 to get the location to write. Then evaluate e_1 to get the value to write. Write the value to the memory at the location. The number of bytes that we write is always the size of the value to write. Return the value of e_1.
- (`create` e ι): Evaluate e to get the number of bytes to allocate. ι is an identifier that is used to identify the object; it is not evaluated. Return the location allocated.
- (`call` $e_0 \cdots e_n$): Evaluate e_0 to get the closure to call. Then, evaluate e_1 up to e_n in order, to get the actual arguments. Bind the actual arguments to the formal arguments and evaluate the body of the closure. The return value size is the size of an integer, by default.
- (`call` $[e_0\ \kappa]\ e_1 \cdots e_n$): Same as the previous case except that the return value size is κ.
- (`op` e_0 e_1): Evaluate e_0 and e_1 in order. Apply the operation and return the result value.
- (`procedure` *Params* *Locals* e): Return a closure of this procedure.
- ι: Return the location that the identifier is currently bound to.
- κ: Return the constant value whose size is the size of the integer, by default.
- $[\kappa_0\ \kappa_1]$: Return the constant value of κ_0 whose size is κ_1 (truncated or extended if necessary).
- $[\iota\ \kappa]$ in declaration: The size of the declared variable ι is κ.

5.1.1 Translation Into MIL

Two things must be mentioned about the MIL programs into which the source programs in C, FORTRAN, or SCHEME are translated.

- Every variable in a MIL program is named uniquely. Parameters, local variables, function names, and the identifiers of `create` expressions[1] are distinct.
- Array and record field access is linearized.

[1]Each malloc site has a unique identifier that will be used to name the dynamically generated objects that come from that site.

```
struct foo {
  int a;
  char b;
};

void main() {
  int a[100][100], i,j, k;
  struct foo *bar;
  struct foo var;

  i = j = k = 0;
  while (1) {
    bar = &var;
    bar->a = 1;
    bar->b = 'a';
    i++;
    j++;
    a[1][90] = 9;
  }
 }
```

Figure 5.2: A C Program

For example, followings are some cases to translate C expressions into MIL correspondents:.

`malloc(`e`)`	$\Longrightarrow$	`(create` e ι`)`	
`&a`	$\Longrightarrow$	`a`	
`= a`	$\Longrightarrow$	`(read a)`	
`a =`	$\Longrightarrow$	`(write a ...)`	
`*a =`	$\Longrightarrow$	`(write (read a) ..)`	
`= *a`	$\Longrightarrow$	`(read (read a))`	
`int a[10]`	$\Longrightarrow$	`(local [a 40])`	
`= a[e]`	$\Longrightarrow$	`(read (+ a` e`))`	
`a[e] =`	$\Longrightarrow$	`(write (+ a` e`) ...)`	
`a->k`	$\Longrightarrow$	`(read (+ a offset))`	$(*)$

Note that in $(*)$, the offset is determined statically by the structure type that `a` points to. An example translation of a C program into its MIL counterpart is shown in Figure 5.2 and Figure 5.3. The while loop in the C program in Figure 5.2 is translated into MIL (Figure 5.3). This front end for C is written by Harrison as a component of the MIPRAC [Har92] compiler system.

5.2 Starting Point: A Template Abstract Interpreter

An abstract interpreter for MIL will be an instantiation of the template abstract interpreter $\mathcal{E}$ in Figure 5.4. Figure 5.4 is in the conventional mathematical format, while Figure 5.2

```
----------------- expressions -----------------

 expr   sym sz al
------------------
  116    30  0  0  (procedure <WHILE-3>
   94        0  0    (if
   39    20  4  4      1
  103        0  0      (begin
   42        4  4        (write
   43    18  4  4          bar
   44    19  4  4          var)
   45        4  4        (write
   46        4  4          (read
   47    18  4  4            bar)
   48    20  4  4          1)
   49        1  1        (write
   50        4  4          (_+
   51        4  4            (read
   52    18  4  4              bar)
   53    21  4  4            4)
   54    22  1  1          97)
   55        4  4        (write
   56    15  4  4          i
   57        4  4          (+
   58        4  4            (read
   59    15  4  4              i)
   60    20  4  4            1))
   61        4  4        (write
   62    16  4  4          j
   63        4  4          (+
   64        4  4            (read
   65    16  4  4              j)
   66    20  4  4            1))
   67        4  4        (write
   68        4  4          (+
   69        4  4            (+
   70    14  4  4              a
   71        4  4              (*
   72    20  4  4                1
   73    23  4  4                400))
   74        4  4            (*
   75    24  4  4              90
   76    21  4  4              4))
   77    25  4  4          9)
  109        0  0        (call
  110        8  8          (read
  111    29  4  4            WHILE-3)))
  112     8  0  0      0))
```

Figure 5.3: While-loop Part of The MIL Version of the Previous C Program

is its Z1 correspondent. We include these two formats in order to help the reader to be accustomed to Z1 expressions.

The template abstract interpreter type is:

$$\mathcal{E}: \Sigma \rightarrow X \rightarrow Y$$

Σ is the set of program points (syntree), X and Y are the pre-state and post-state lattices, respectively. Depending on how we define the pre-state lattice X, the post-state lattice Y, and the template functions, different abstract interpreters are instantiated. Every abstract interpreter in this chapter is an instantiation of this template abstract interpreter.

The functions used in the template abstract interpreter hide the details of the evaluation rules, which are dependent on the pre-state and post-state lattice definitions.

- **Next**. Sequential evaluation of expressions uses the post state of the previous expression evaluation. Because, in general, the pre-state and the post-state is not identical, the post-state of the previous evaluation cannot directly be the pre-state for the next evaluation. **Next** selects appropriate components from the post state of the previous expression evaluation and prepares the pre state for the next expression evaluation.
- **AtLeastTrue** and **AtLeastFalse**. Test whether the abstract condition value is true (respectively, false) or a "don't-know" value.
- **LocationValue** and **ClosureValue**. An abstract value can be an abstract location, an abstract integer, or an abstract closure. **LocationValue** (respectively, **ClosureValue**) selects the abstract location (respectively, closure) portion of an abstract value.
- **Fetch**. Fetch the abstract value in a location of the abstract store.
- **Define**. Write an abstract value into a location of the abstract store.
- **Apply**. Apply an abstract closure to the abstract argument values.
- **PlusOp**. Apply the binary operation to the two abstract operands.
- **Allocate**. Allocate a chunk of memory in the abstract store.
- **Closure**. Create an abstract closure value.
- **Location**. Create an abstract location value.
- **Constant**. Create an abstract integer value.

The Z1 version, `Eval`, of the template abstract interpreter E is shown in Figure 5.2.

$$
\begin{array}{lll}
\mathcal{E}\colon \Sigma \to X \to Y & & \\
\mathcal{E}(\sigma, x) = \ \text{case } \sigma \text{ of} & & \\
(\texttt{begin } e_1 \cdots e_n) : & \text{let} & y_1 = \mathcal{E}(e_1, x) \\
 & & y_i = \mathcal{E}(e_i, \mathbf{Next}(x, y_{i-1})) \qquad \text{for } 2 \le i \le n \\
 & \text{in} & y_n \\
(\texttt{if } e_1 \ e_2 \ e_3) : & \text{let} & y_1 = \mathcal{E}(e_1, x) \\
 & & v = \mathbf{BooleanValue}(y_1) \\
 & & y_2 = \text{if}(\mathbf{AtLeastTrue}(v)) \text{ then } \mathcal{E}(e_2, \mathbf{Next}(x, y_1)) \text{ else } \bot \\
 & & y_3 = \text{if}(\mathbf{AtLeastFalse}(v)) \text{ then } \mathcal{E}(e_3, \mathbf{Next}(x, y_1)) \text{ else } \bot \\
 & \text{in} & y_2 \sqcup y_3 \\
(\texttt{read } e) \text{ or } (\texttt{read } [e \ \kappa]) & & \\
 & \text{let} & y_1 = \mathcal{E}(e, x) \\
 & & l = \mathbf{LocationValue}(y_1) \\
 & & y_2 = \mathbf{Fetch}(l, y_1, \kappa) \qquad \kappa \text{ is the integer size when unspecified} \\
 & \text{in} & y_2 \\
(\texttt{write } e_1 \ e_2) : & \text{let} & y_1 = \mathcal{E}(e_1, x) \\
 & & y_2 = \mathcal{E}(e_2, \mathbf{Next}(x, y_1)) \\
 & & l = \mathbf{LocationValue}(y_1) \\
 & & y_3 = \mathbf{Define}(l, y_2) \\
 & \text{in} & y_3 \\
(\texttt{call } e_1 \cdots e_n) \text{ or } (\texttt{call } [e_1 \kappa] \cdots e_n) : & & \\
 & \text{let} & y_1 = \mathcal{E}(x) \\
 & & y_i = \mathcal{E}(e_i, \mathbf{Next}(x, y_i)) \qquad \text{for } 2 \le i \le n \\
 & & c = \mathbf{ClosureValue}(y_1) \\
 & & y_{n+1} = \mathbf{Apply}(c, y_2, \cdots, y_n, \kappa) \quad \kappa \text{ is the integer size when unspecified} \\
 & \text{in} & y_{n+1} \\
(\texttt{+ } e_1 \ e_2) : & \text{let} & y_1 = \mathcal{E}(e_1, x) \\
 & & y_2 = \mathcal{E}(e_2, \mathbf{Next}(x, y_1)) \\
 & & y_3 = \mathbf{PlusOp}(y_1, y_2) \\
 & \text{in} & y_3 \\
(\texttt{create } e \ \iota) : & \text{let} & y_1 = \mathcal{E}(e, x) \\
 & & y_2 = \mathbf{Allocate}(\iota, y_1) \\
 & \text{in} & y_2 \\
(\texttt{procedure ...}) : & \mathbf{Closure}(e, x) & \\
\iota : & \mathbf{Location}(e, x) & \\
\kappa : & \mathbf{Constant}(e, x) &
\end{array}
$$

Figure 5.4: The Template Abstract Interpreter in Conventional Mathematical Form

```
(Eval (from syntree X) (to Y))
(semantics Eval (expr x)
  (switch
    (isBegin expr)
    (begin
      (let (_ y 1) (Eval (nthsubexpr expr 1) x)
           (_ y i) (for i 2 (numofsub expr)
                         (Eval (nthsubexpr expr i) (Next x (_ y (- i 1))))))
      (_ y (numofsub expr)))

    (isIf expr)
    (begin
      (let y1 (Eval (nthsubexpr expr 1) x))
      (let v (BooleanValue y1))
      (let y2 (if (AtLeastTrue v)
                  (Eval (nthsubexpr expr 2) (Next x y1))
                  (bottom Y)))
      (let y3 (if (AtLeastFalse v)
                  (Eval (nthsubexpr expr 3) (Next x y1))
                  (bottom Y)))
      (join y2 y3))

    (isRead expr)
    (begin
      (let y1 (Eval (nthsubexpr expr 1) x))
      (let loc (LocationValue y1))
      (let y2 (Fetch loc y1))
      y2)

    (isWrite expr)
    (begin
      (let y1 (Eval (nthsubexpr expr 1) x))
      (let y2 (Eval (nthsubexpr expr 2) (Next x y1)))
      (let loc (LocationValue y1))
      (let y3 (Define loc y2))
      y3)

    (isCall expr)
    (begin
      (let (_ y 1) (Eval (nthsubexpr expr 1) x)
           (_ y i) (for i 2 (numofsub expr)
                         (Eval (nthsubexpr expr i) (Next x (_ y (- i 1))))))
      (let c (ClosureValue (_ y 1)))
      (let y (Apply c (_ y 2) ... (_ y (numofsub expr))))
      y)

    (isPlus expr)
    (begin
      (let y1 (Eval (nthsubexpr expr 1) x))
      (let y2 (Eval (nthsubexpr expr 2) (Next x y1)))
      (let y3 (PlusOp y1 y2))
      y3)

    (isCreate expr)    (Allocate expr x)
    (isProcedure expr) (Closure expr x)
    (isVar expr)   (Location expr x)
    (isConst expr) (Constant expr x)
  ))
```

The three-dot expression in the body of `isCall` branch is not a legitimate Z1 expression. All the index variables `(_ y i)` should explicitly be passed to `Apply`.

Figure 5.5: The Template Abstract Interpreter in Z1

5.3 Experiment Snapshots

The following are snapshots of our experiment steps. We execute the analyzer inside `GDB`[2] because there we can examine the analysis output (a binary image) interactively.

Figure 5.6 shows the analysis of an input program `alias.ifi` by the generated analyzer. The analysis result (a binary image) is dumped into `alias.ifi.dmp`. In order to examine the analysis result, we need to read back the dumped fixpoint and probe its entries. The dumped fixpoint consists of two tables `Tx` and `T` that map each expression point to its pre-state and post-state, respectively. The analyzer has a `-Q` (query) option. Re-running the analyzer with `-Q` option causes it to read back the dumped fixpoint result `alias.ifi.dmp`. Inside `GDB` we set a break point so that after the analyzer reads back the dumped fixpoint it does not exit. For this purpose, system Z1 generates a dummy procedure `Query` in the analysis source. We set break point at `Query`, re-run the analyzer with `-Q` option. The analyzer reads in the dumped fixpoint result, stops at the dummy procedure entry `Query`, and waits for our `GDB` commands to probe the fixpoint values. Figure 5.7 shows this process.

Now, we can issue `GDB` commands to examine the fixpoint values. For this purpose system Z1 generates, for each type `<T>`, a pretty-printer routine named `Display<T>`. Type of each display routines `Display<T>` is

```
Display<T>(<T>, FILE*, int)
```

The first argument is the element to display, the second the stream pointer (`stdout, stderr`) and the third the indentation count (non-negative number). Using these routines we can observe abstract in readable form. The pre-state (respectively post-state) for a program point n is in `Tx[`n`]` (respectively `T[`n`]`). For example, inside `gdb`,

```
(gdb) print DisplayPRE(Tx[10],qout,0)
```

will show the pre-state value (whose type name is `PRE`) at the program point 10. Output from the display routine is in the following format. The first column is the name of a type.

- A function domain element f is printed as

  $$[x_0 \texttt{ -> } f(x_0)]\cdots[x_n \texttt{ ->} f(x_n)]$$

  for all x_i such that f is defined at x_i.

- A powerset domain element is printed in enumerated set notation:

  $$\{\texttt{ a, b, c}\}$$

[2]GNU symbolic debugger 4.2

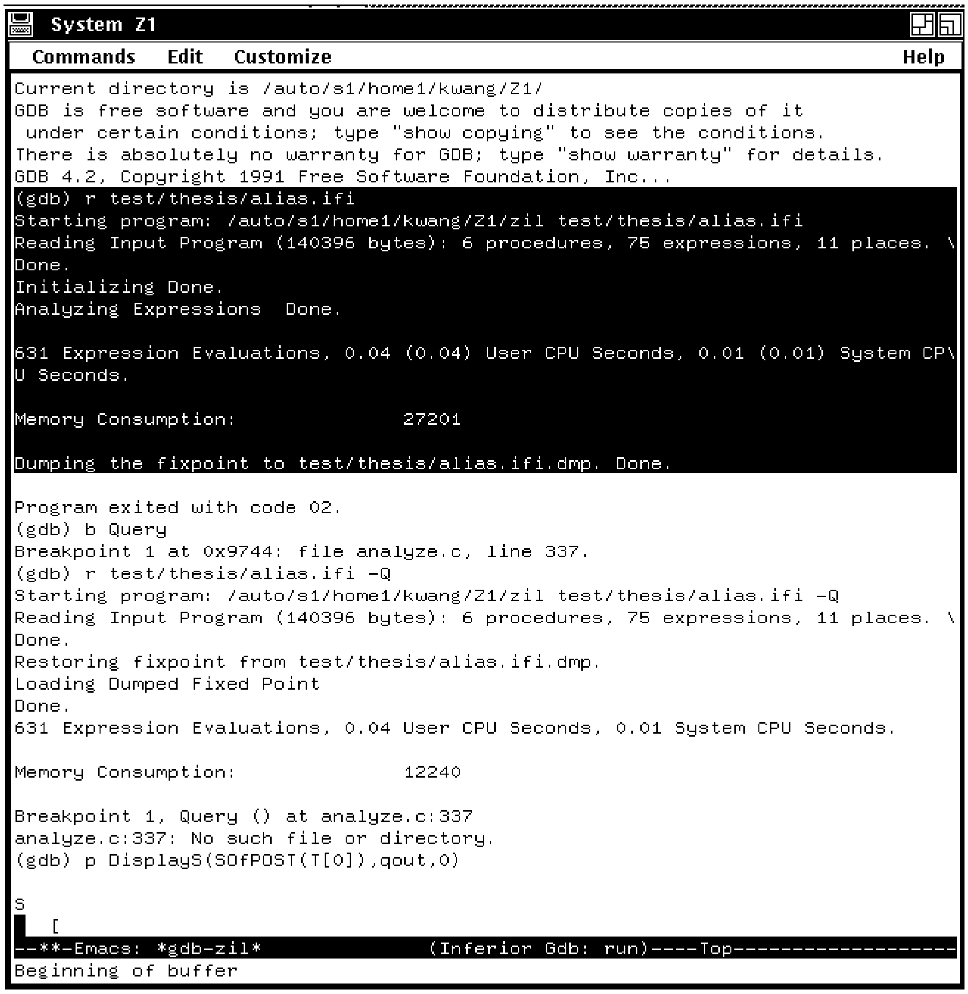


Figure 5.6: Running the Analyzer with Source Program `alias.ifi`

Figure 5.7: Setting a Break Point at `Query` and Re-run the Analyzer with `-Q` Option

- A product domain/set element x is printed as

$$<x \downarrow 0, \ldots, x \downarrow n>$$

- The bottom (respectively top) element is printed as _ (respectively ^).

Probing the post-state `T[0]` at the program point 0 is shown in Figure 5.8.

5.4 Alias Analysis and Constant Propagation

The abstract interpreter for alias analysis and constant propagation is a starting point from which other abstract interpreters in this chapter are derived by adding some extra mechanisms. The abstract interpreter for constant propagation and alias analysis does not require an extra instrumentation to the standard (but abstract) interpretation rules of the target language.

Definition 23 (Alias) During a program execution, when two different expressions evaluate to the same location value l, they are aliases for l.

Depending on the target languages, the above definition of aliases can be overly general. This is because some languages allow only a restricted set of aliases. For example, in Fortran, aliases can occur only through the `EQUIVALENCE` variables and reference formal parameters. For Fortran programs it is enough to compute the variable pairs that represent the same memory location. However, in C, reference parameters are not the only source of alias relations. In C, unrestrained pointer manipulations can be applied to arbitrary memory locations. Thus, alias analysis for C programs requires us to expand the conventional definition of alias (between variable names) to a more general one like Definition23. Consider a C program in Figure 5.9. When `foo` is active, `x.array` and the integer array variable `a` are aliases. `(x.array)+10` and the l-value of `a[10]` are aliases to the tenth element of the array.

Our alias analysis computes an approximate location for each location expression. Alias relations are determined as follows:

> For each pair of location expressions e_1 and e_2, if their values v_{e_1} and v_{e_2} have the same abstract location value, then the two expressions may access to the same memory location for some execution instances of the program.

Note that the analysis computes the *approximate may-alias* relations. It is approximate because an abstract location denotes a set of concrete locations. Thus, when two expressions evaluate to the same abstract location, it does not necessarily mean that the two expressions

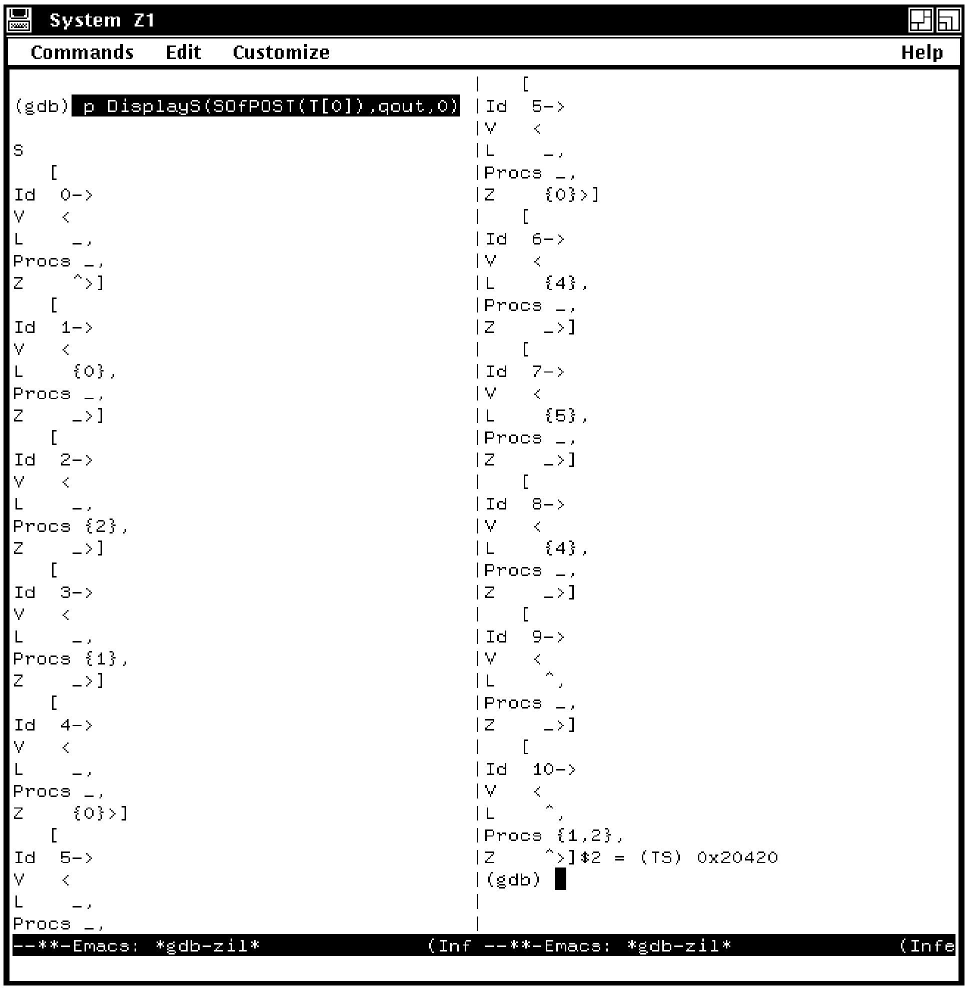


Figure 5.8: Probing the Post-state Value at Expression Point 0

```
void foo() {
   int a[100];
   struct {int *array; int index} x;

   x.array = a;
   ... *((x.array) + 10) ...
   ... a[10] ...
}
```

Figure 5.9: Example C Expressions That Are Aliases

are aliases to the same location in reality. We guarantee, however, that a superset of the real alias relations are computed. It computes may-aliases: that expressions e_1 and e_2 evaluate to the same abstract location means that for some (not necessarily all) execution instances of the program they may be aliases. In other words, the only guarantee is that if they are not abstract aliases, they are not concrete aliases.

Alias analysis by abstract interpretation requires an abstract location domain. Similarly, constant propagation requires an abstract integer domain.

Definition 24 (Constant Propagation) During program execution, some expressions evaluate to constant integer values. Constant propagation computes which expressions remain as constants for all execution instances.

Conventional constant propagation focuses only on the program variables. Our constant propagation analysis finds out the expressions (not restricted to the program variables) that evaluate to constant integers.

Note that an abstract interpreter for constant propagation and alias analysis does not need an extra mechanism for recording non-standard data values. Locations and integers are among the standard data values. We only need an abstract version of the standard interpreter that uses abstract locations and abstract integers.

We present the abstract interpreter specification in Z1. Our specification is an instantiation of the template abstract interpreter definition in page 76. The abstract interpreter `Eval` is of the type:

```
(Eval (from syntree PRE) (to POST))
```

Given a program point (of type `syntree`) and a pre-state ($\in$ `PRE`) at that point, `Eval` evaluates the expression at that point and generates a post-state ($\in$ `POST`).

In this section, we will define three abstract interpreters **Eval1**, **Eval2**, and **Eval3**. **Eval1** is the most primitive abstract interpreter among them, while **Eval3** is the most elaborate one. We will demonstrate, by these three abstract interpreters, that the Z1 permits us to define elaborate and highly accurate abstract interpreters.

- **Eval1**: The first interpreter **Eval1** is the most primitive one in the sense that it abstracts distinct instances of a program variable into single one, and consider all array and record elements as a single memory location. Thus, during abstract interpretation, writing a value into the abstract store is merely moving up a lattice chain of the stored values, instead of overwriting previous values. Consider the locations bound with to the local variables of a recursive procedure. At every recursive call to the procedure, distinct instances of its local variables are allocated on the stack. During the abstract interpretation of **Eval1**, however, all these distinct instances are given a single location in the abstract store. Moreover, consider an array of, say, 10 elements. Distinct elements are accessed by distinct array indexing expressions. The abstract store in **Eval1**, however, has only one abstract location for an array; its distinct elements share the same slot in the abstract store. Similarly, distinct fields of a record variable share a single slot (named by the record variable) of the abstract store.

- **Eval2**: The second interpreter **Eval2** adds to **Eval1** an abstract mechanism for differentiating variable instances.
- **Eval3**: The third interpreter **Eval3** adds to **Eval2** an abstract mechanism for differentiating the elements of an array. It also differentiates the fields of a record variable.

5.4.1 Eval1

The pre-state `PRE` is simply a store `S`.

```
(domain PRE (synonym S))
```

The post-state `POST` is a pair of a value `V` and a store `S`

```
(domain POST (* V S))
```

The first component `V` is the value to which the current expression evaluates. The second component `S` is the store after the evaluation of the current expression. The store is a function lattice from an identifier `Id` to its value `V`.

```
(domain S (-> Id V))
```

Because we do not differentiate the instances of a variable, the store maps a variable name directly to its value. There is only one abstract location for a variable. Distinct instances of a variable share a single slot of the store. This can have a detrimental effect on the analysis accuracy because we cannot overwrite values in the store. For example, the local variables of a recursive procedure will be instantiated several times on the stack during execution. In **Eval1**, assigning a new value to one of these instances will be simulated by joining with the previous value value instead of overwriting it. One way to differentiate the instances of an identifier is to make the abstract store a two-level map; it maps an identifier to a function that maps an instance to its value. We will present an instance-sensitive abstract store shortly in Section 5.4.2.

The value of an expression can be a number `Z`, a location `L`, or a closure `C`

```
(domain V (* L C Z))
```

An abstract location is a collection of identifiers `setId`.

```
(domain L (2^ setId))
```

The set `setId` $= \{0, \cdots, \iota - 1\}$ of the identifier indices in the source program is determined by the number `numOfPlaces` of the identifiers in the program.

```
(set setId (index NumOfPlaces)).
```

The abstract integer is a powerset lattice from a finite set `setZ` of integers.

```
(domain Z (2^ setZ))
(set setZ (range -999999 999999))
```

Note that the set `setZ` of integers is a finite set bounded by -999999 and 999999. When a program produces an integer value beyond this range we substitute the top integer value for it. The auxiliary function `numToZ` is used to check this case. Note that the abstract integer space is a very rich structure for constant propagation. Our plan is that, at the outset, we define a very detailed lattice structure for abstract integers and abstract locations. We then apply projections to these lattices in order to simplify their structures when necessary. For example, for constant propagation analysis we may wish to project the powerset lattice into the flat one by means of

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

By using different projections we can achieve distinct constant propagation analyses which vary in their analysis capability and cost. Sometimes the projections of the integer lattice structure also affect the accuracy of the alias analysis. This is because when we are able to compute an accurate integer value some spurious aliases relations can be avoided. Suppose two different aliases are captured in the two branches of an if expression. Suppose that the condition expression's value is affected by the integer lattice structure. If we used a detailed integer lattice so that we could determine which branch the execution follows (instead of following both branches and joining later), this would eliminate the otherwise spurious alias relations caused by following both branches.

The abstract closure is a collection of procedure indices. The closure value does not have environment component. This is because every identifier is distinct and because we do not differentiate the instances of an identifier.

```
(domain C (synonym Procs))
(domain Procs (2^ setProc))
```

The set `setProc` $= \{0, \cdots, n - 1\}$ of procedure indices is determined by the source program, where n is the number of procedures ($n =$ `numOfProcs()`) in the program.

```
(set setProc (index NumOfProcs))
```

In summary, the lattice definitions of the abstract interpreter **Eval1** are in Figure 5.10.

Before we present the body of the abstract interpreter, let us first present some auxiliary functions that are used inside it. We will show only their types and will explain what they do. The reader will find their complete definitions in Appendix B.

```
(types
 (domain PRE (synonym S))     ; pre-state is the store
 (domain POST (* S V))        ; post-state is a pair of the store and value
 (domain S (-> Id V))         ; store is a function from id to value
 (domain V (* L C Z))         ; value is a loc, closure, or integer
 (domain L (2^ setId))        ; location is a set of ids
 (domain C (synonym Procs))   ; closure is a set of proc ids
 (domain Z (2^ setZ))         ; abstract integer
 (domain T (flat setB))       ; abstract boolean
 (domain Ids (2^ setId))
 (domain Id (flat setId))
 (domain Procs (2^ setProc))
 (domain Proc (flat setProc))
 (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 integers
)
```

Figure 5.10: Abstract Domains for **Eval1**

First, some foreign procedures that access the abstract syntax tree of the source program. The interpretation of a language construct may require to access some syntactic objects associated with the construct. All foreign procedures are defined in C. The following foreign procedures access the syntax tree of the source program and returns value that we need during the interpretation.

- `(NumOfExprs NumOfProcs NumOfPlaces (from) (to number))`

  `NumOfExprs` returns the number of expressions of the input program. `NumOfProcs` returns the number of procedures in the input program. `NumOfPlaces` returns the number of identifiers in the input program. Every identifier in the input MIL program is distinct.

- `(NumOfSub (from syntree) (to number))`: returns the number of subexpressions of an expression.

- `(NumOfParam (from setProc) (to number))`: returns the number of parameters of a procedure.

- `(NumOfLocal (from setProc) (to number))`: returns the number of local variables of a procedure.

- `(NthSubexpr (from syntree number) (to syntree))` and
  `(NthParam NthLocal (from setProc number) (to setId))`.

  `(NthSubexpr x n)` returns the `n`-th subexpression of an expression `x`. `(NthParam p n)` and `(NthLocal p n)` return the `n`-th parameter and local variable of a procedure `p`.

- `(Owner (from setId) (to setProc))`

  `(Owner i)` returns the procedure index which has `i` as its local variables or parameter. Every identifier in a MIL program is distinct, thus its owner is unique.

- `(DynamicId (from syntree) (to setId))`

  Each `create` expression is named uniquely so that the allocated memory has an entry for the store map. `(DynamicId x)` returns that identifier index associated with the `create` expression `x`.

- `(ProcId (from syntree) (to setProc))`

  Each function in a MIL program has a unique name. Like a `create` expression, a closure expression (lambda expression) is associated with its name, a procedure index. `(ProcId x)` returns the procedure index associated with the closure expression `x`.

- `(Xid (from syntree) (to setId))`: returns the identifier index for a variable expression.

- `(Xnum (from syntree) (to number))`: returns the numeric value of a constant expression.

- `(Body (from setProc) (to syntree))`: returns the body of a procedure.

- `(isBegin isIf isRead isWrite isCreate isCall isApply isProc isClosure isId isNumber isPlus isMinus isMulti isMod isDiv isEq isLe isLt isGe isGt isNot isAnd isOr isEtcBiOp isEtcUnOp (from syntree) (to number))`

  The above functions test the syntactic construct of an expression. For example, `(isBegin x)` returns 1 if the expression `x` is a begin expression.

Now, the non-foreign auxiliary functions. These are functions whose definitions are written directly in Z1.

- `(numToZ (from number) (to Z))`

  A constant integer appearing in the program text is translated into an abstract integer by `numToZ`.

- `(· (from Z Z) (to Z))`

  Abstract binary operators over the abstract integer `Z` are defined by `plusZ`, `minusZ`, `multiZ`. "·" may therefore be any of `plusZ`, `minusZ`, or `multiZ`.

- `(· (from Z Z) (to T))`

  Abstract relational operators over the abstract integer are defined. "·" is one of `leZ`, `ltZ`, `geZ`, or `gtZ`.

- `(· (from T T) (to T))`

  Abstract boolean operators over the abstract boolean values are defined. "·" is one of `andT`, `orT`, or `notT`.

- `(numTosetB (from number) (to setB))`

  Sometimes we need to change a number (not an abstract integer value) into a boolean value.

- `(ZtoT (from Z) (to T))` and `(TtoZ (from T) (to Z))`

  An abstract integer value and an abstract boolean value may be used interchangeably. `ZtoT` changes an abstract integer into the corresponding abstract boolean value, and `TtoZ` makes the same conversion in the other direction.

- `(equalL (from L L) (to T))` and `(equal C (from C C) (to T))`

  These check the equality of abstract locations and closures.

Now the semantics function `Eval`. For each language construct of MIL, `Eval` defines the abstract evaluation rule.

- For the `begin` expression, it evaluates the sequence of subexpressions and return the last post-state as the post context of the `begin` expression.

```
(isBegin x)
(begin
 (let (_post 1) (Eval (NthSubexpr x 1) s)
      (_post i) (for i 2 (NumOfSub x)
                    (Eval (NthSubexpr x i) (! (_post (- i 1)) S)
                          )))
 (_post (NumOfSub x))
 )
```

- For the `if` expression, it evaluates the condition expression. Depending on the abstract condition value, it evaluates the true branch, the false branch, or both branches.

```
(isIf x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 S))
 (let v1 (! post1 V))    ; get the condition value
 (let cond (equalZ (! v1 Z) (Z (setZ 0)))) ; see if v1 is euqal to 0
 (let ppost2 (if (<= (T False) cond)
                 (Eval (NthSubexpr x 2) s1) ; follow true branch
               (POST bottom)))
 (let ppost3 (if (<= (T True) cond)         ; follow false branch
                 (Eval (NthSubexpr x 3) s1)
               (POST bottom)))
 (join ppost2 ppost3)
 )
```

- For a read expression, it reads the store entry at the abstract location given by the first subexpression. Note that an abstract location is a set of identifiers. The value stored at that abstract location is the join of the store entries for the identifiers that constitute the abstract location.

```
(isRead x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 S))
 (let loc :Ids (! (! post1 V) L))                ;location value
 (let v1 (mapjoin (lambda (from setId) (to V) (i)
                   (@ s1 (Id i)))
                 loc))
 (POST (! post1 S) v1)
 )
```

- For a write expression, it evaluates the abstract location (the value of the first subexpression) and the abstract value to write (the value of the second subexpression). It updates the store with the new value. The update cannot overwrite the old value, since our abstract store entry represents more than one concrete location.

```
(isWrite x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let loc :Ids (! (! post1 V) L))       ; location to write
 (let post2 (Eval (NthSubexpr x 2) (! post1 S)))
 (let s2 (! post2 S))
 (let v2 (! post2 V))      ; value to write
 (let s3 (mapjoin          ; join the new store image with the old one
          (lambda (from setId) (to S) (id)
            ([/] s2
                 (by (Id id)
                     (join (@ s2 (Id id)) v2))))
          loc))
 (POST s3 v2)
 )
```

- For a create (memory allocation) expression, there is no change to the store. The value is an abstract location value, which is the identifier associated with the create expression.

```
(isCreate x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1  (! post1 S))
 (let id  (DynamicId x))    ; identifier index associates with x
 (let v (V (Ids id) (C bottom) (Z bottom)))
 (POST s1 v)
 )
```

- For a procedure generating expression, there is no change to the store. The value is an abstract closure.

```
(or (isProc x) (isClosure x))
(POST s (V (L bottom)
           (C (Procs (ProcId x)))
           (Z bottom)))
```

- For an identifier expression, it returns the abstract location value bound to the identifier.

```
(isId x)
(begin
 (let id (Xid x))
 (POST s (V (Ids id) (C bottom) (Z bottom)))
 )
```

- For a number expression, it returns the abstract integer value. We need to check that the number is within the range of `setZ`; this check is done by `numToZ`.

```
(isNumber x)
(POST s (V (L bottom) (C bottom) (numToZ (Xnum x))))
```

- For a procedure call expression, it evaluates the arguments, puts their values in the locations to which the formal parameters are bound, initializes the local variables, and evaluates the body of the abstract procedure that is called. The abstract procedure is a collection of procedure indices. For each procedure we apply its body and join the final stores after control returns from the procedures.

```
(or (isCall x) (isApply x))
(begin
 (let n (NumOfSub x))
 (let num_actual (- n 1))
 ;
 ; evaluate the closure value and the arguments
 ;
 (let (_post 1) (Eval (NthSubexpr x 1) s)
      (_post i) (for i 2 n
                     (Eval (NthSubexpr x i)
                           (! (_post (- i 1)) S))))
 ; get the closure value
 (let (_v i) (for i 1 n (! (_post i) V)))
 (let s1 (! (_post n) S))
 (let procs (! (_v 1) C))
 ;
 ; for each procedure index represented by the abstract closure value
 ;
 (mapjoin
  (lambda (from setProc) (to POST) (alpha)
     (begin
     (let mom (Body 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)))))
        ;
        ; parameter binding
        ;
        (let (_s2 0) s1
             (_s2 i) (for i 1 num_param
                          (begin
                           (let param :setId (nth params i))
                           ([/] (_ s2 (- i 1))
                                (by (Id param)
                                    (join (@ (_ s2 (- i 1)) (Id param))
                                          (_ v (+ i 1)))))
                           )))
        (let num_local (NumOfLocal alpha))
        (let locals :Ids (join (for i 1 num_local
                             (Ids (NthLocal alpha i)))))
        ;
        ; initialized the local variables
        ;
        (let (_s3 0) (_s2 num_param)
             (_s3 i) (for i 1 num_local
                          (begin
                           (let local :setId (nth locals i))
                           ([/] (_s3 (- i 1))
                                (by (Id local)
                                    (join (@ (_ s3 (- i 1)) (Id local))
                                          (V (L bottom)
                                             (C bottom)
                                             (Z (setZ 0))))))
                           )))
        ;
        ; evaluate the body of the procedure
        ;
        (Eval mom (_ s3 num_local))
        ))
     ))
    procs
    ))
```

- For each binary operation expression, it applies the abstract version over the abstract integers and/or abstract boolean values. One should note that the plus operation must handle the addition of location values with integer values, because this is the way that array indices and structure offsets are expressed. For example,

  `(+ x 4)`

  is evaluated into the location which is offset by 4 bytes from the location bound to the variable `x`. However, since `Eval` uses a primitive store model that has only one abstract location for array and structure variables, the operation '+' rule does not contain the location addition rules. It merely joins the location values that are returned by the two operands.

```
(isPlus x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
```

```
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (join (! v1 L) (! v2 L))
           (C bottom)
           (plusZ (! v1 Z) (! v2 Z))))
 (POST s2 v)
 )

(isMulti x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
           (C bottom)
           (multiZ (! v1 Z) (! v2 Z))))
 (POST s2 v)
 )
```

All other cases are similarly defined.

- For an expression which escapes our evaluation rule net, we just return the top integer value. This is included only because the syntax of the `switch` expression requires it. There should not be any case that reaches this branch.

```
default   ; return top integer value
(POST s (V (L bottom) (C bottom) (Z top)))
```

The complete specification of **Eval1** is in Appendix B. From this specification system Z1 generates a C code for the analysis. After we compile and link the C code with the library and MIL parser, it become an executable analyzer.

We will experiment with several projections for **Eval1**. We will apply projections to two lattices: to the abstract integer domain

`(domain Z (2^ setZ)),`

in order to see the change in accuracy for constant propagation, and to the abstract location domain

`(domain L (2^ setId)),`

in order to see the difference in accuracy for alias analysis.

Consider the C program in Figure 5.11.

• Analysis of `alias.c` (Figure 5.11) With Projections `(project Z (height> 1))` and `(project L (height> 1))`

```
#define UNKNOWN (-0x8000000)

void f(int *x, int *y) {
  int m;
  if (UNKNOWN) {
    m = *x;
    f(x,x);                    (1)
  }
  else m = *y;
}

void main () {
  int i, j;
  int *a, *b;

  i = j = 0;
  a = &i;                      (2)
  b = &j;                      (3)
  f(a,b);                      (4)
}
```

Figure 5.11: C Program `alias.c`

By `(project Z (height> 1))`, the abstract integer lattice becomes the flat lattice. Similarly, the abstract location lattice becomes the flat lattice. Therefore, any expression that evaluates to more than one integer value is projected to the top integer. By the same token, any expression whose value has more than one location value is projected into top location by the projection `(project L (height> 1))`.

Consider `alias.c` (Figure 5.11). Inside the `main` procedure, variables `a` and `b` are assigned the locations of variables `i` and `j`, respectively (line 3 and 4). These locations are passed to procedure `f` (line 5).

We examine the store component of the post-state value associated with the last program point. Figure5.12 shows that the variables `a` (index 6) and `b` (index 7) contain location 4 (variable `i`) and 5 (variable `j`), respectively.

The formal parameters `x` (Id 8) and `y` (Id 9) of `f` point to the locations of the main procedure's local variables `i` (Id 4) and `j` (Id 5), respectively. Because of the recursive call `f(x,x)` (line 1) inside `f`, `y` can also point to the location pointed by `x`. Since the location lattice is projected into the flat lattice (by `(project L (height> 1))`) from `(domain L (2^ setId))`, this alias information of `y` is represented as the top location (Figure 5.13). This would have a detrimental effect on the analysis accuracy, because the read expression (line 2) that accesses the location bound to `y` should read all the entries (the top location) of the store and return a value that is safely related to all of them.

- **Analysis of `alias.c` (Figure 5.12) with Projection `(project Z (height> 1))` and**

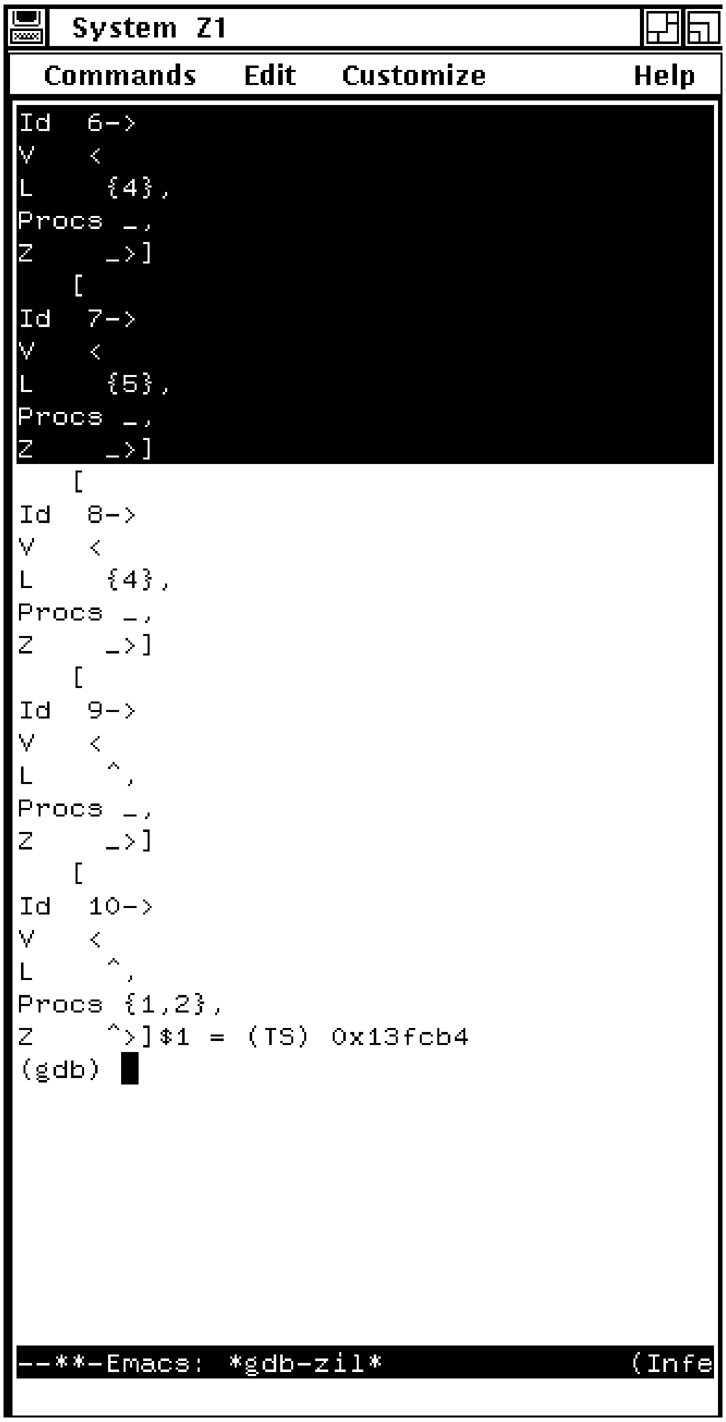


Figure 5.12: Analysis Result for `a` and `b`

```
System Z1
 Commands   Edit   Customize          Help
Id  6->
V   <
L    {4},
Procs _,
Z    _>]
   [
Id  7->
V   <
L    {5},
Procs _,
Z    _>]
   [
Id  8->
V   <
L    {4},
Procs _,
Z    _>]
   [
Id  9->
V   <
L    ^,
Procs _,
Z    _>]
   [
Id  10->
V   <
L    ^,
Procs {1,2},
Z    ^>]$1 = (TS) 0x13fcb4
(gdb)

--**-Emacs: *gdb-zil*              (Infe
```

Figure 5.13: Analysis Result For x And y With Flat Location Lattice

No Projection of Locations

Let's analyze the same program `alias.c` with different projections. We found that the flat location lattice would result in poor accuracy. An expression easily evaluates to the top location, which causes a later read expression to access all the entries of the store, join them, and return this result as a value of the read expression. Now let us consider the outcome if we avoid projecting the abstract location domain; that is, it remains a powerset lattice.

At the procedure call statement (line 5), the formal parameters `x` and `y` of `f` point to the locations bound to the main procedure's local variables `i` and `j`, respectively. Because of the recursive call `f(x,x)` (line 2) inside `f`, `y` can also points to the location bound to `x`. Because the location lattice is a powerset lattice without any projection, the second formal parameter `y` of `f` contains the locations of `i` (Id 4) and `j` (Id 5), instead of having the top location as way previously the case. The analysis result shows this (Figure 5.14).

Note that because **EvalI** is insensitive to the distinct instances of variables, it cannot detect that the first instance of `y` from the initiating call to `f` in line 5 contain the location of `j`, while the other instances by the recursive calls to `f` in line 2 contain the location of `i`. It can show only that the parameter `y` contains the location of either `i` or `j`. If we differentiated the variable instances between the one allocated by the initiating call to the binding procedure and those allocated by subsequent recursive calls, the analysis would have been able to detect that the instance of `y` by the initiating call to `f` (line 5) points to `j` and other instances of `y` allocated by the recursive calls (line 2) point to `i`. In Section 5.4.2 we will define an abstract interpreter that uses an instance-sensitive store lattice.

• Analysis of `alias2.c` (Figure 5.15) with Projection `(project Z (height> 1))` and No Projection of Locations

Now consider the C program of Figure 5.15. By this example, we will show that the accuracy of alias analysis can be affected by the projections of the abstract integer lattice. The first formal parameter `m` of `g` points either to `i` or to `j` depending on the if condition at line 6. Similarly, its second parameter `n` points either to `i` or `j`. Note that inside the `for` expression (line 4) `f` assigns 1 (line 1) or 0 (line 2) to the location pointed to by its parameter `x`. If we could detect this, we could evaluate only one branch at the `if` expression at line 6. Depending on the projections over the abstract integer lattice, this is possible and can help to improve the accuracy of the analysis.

Let's first consider the flat integer lattice that results from `(project Z (height> 1))`, and no projection of the location lattice. See Figure 5.16 for the analysis result for `x`. The value of `x` (Id 8) is the top integer; because the integer lattice is flat, the fact that it has two different values (1 or 0) is represented by top. Because of this, the `if` condition expression evaluates to a "don't-know" value. Hence both branches of the `if` expression are followed and their post-states are joined. This causes the alias information of the two

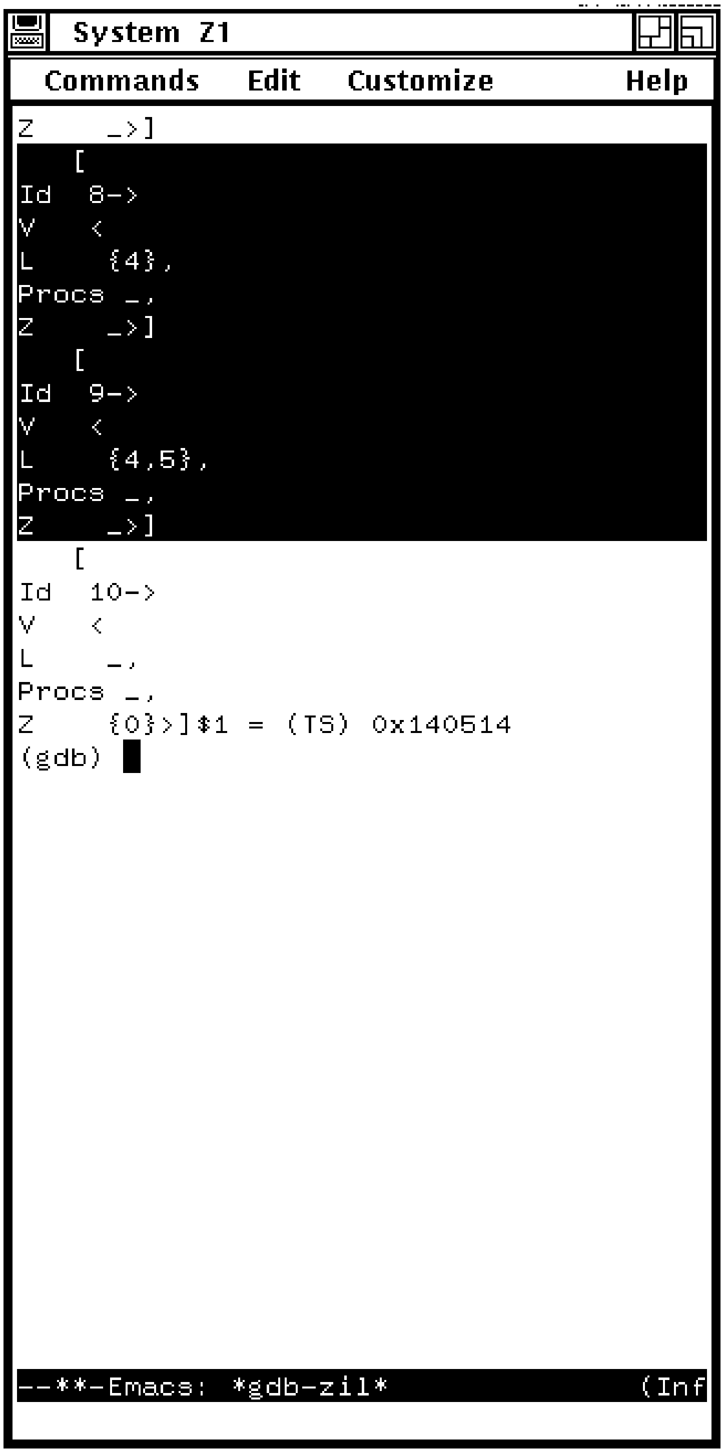


Figure 5.14: Analysis Result For x And y Without Projection For Location Lattice

```
void f(int *x) {
  if (*x <= 0) *x = 1;              (1)
  else *x = 0;                      (2)
}

void g(int *m, int *n) {
  *m = *n+1;                        (3)
}

void main () {
  int i, j, k, x;
  int *a, *b;

  i = j = 0;
  a = &i;
  b = &j;
  for (k=0; k<100; k++)              (4)
    f(&x);                           (5)
  if (x >= 0) g(a,b);                (6)
  else g(b,a);                       (7)
}
```

Figure 5.15: C Program `alias2.c`

formal parameters `m` (Id 12) and `n` (Id 13) of `g` to become vague. See Figure 5.17. It shows that `m` contains the location of either `i` (Id 5) or `j` (Id 6), and similarly for `n`.

• Analysis of `alias2.c` (Figure 5.15) with Projection `(project Z (height> 2))` and No Projection of Locations

Now, let's use a different projection for the abstract integer domain. Instead of making the powerset lattice into the flat one, we make it such that two-valued integers exist. The projection

`(project Z (height> 2))`

does the job. Singleton integer sets and two-element integer sets are still elements of the projected lattice, but integer sets that contain more than two elements are projected into top. The new analyzer's analysis output is as we expected. Note that the value of `x` (Id 8) is indicated as 1 or 0 (Figure 5.18). And the formal parameter `m` (Id 12) of `f` contains only the location of `i` (Id 5), because the false branch of the `if` expression at line 6 is determined to be dead code (the if condition is always true).

• Analysis of `cp.c` (Figure 5.20) with Projections `(project Z (height> 1))` and `(project L (height> 1))`

We now focus on constant propagation. Firstly, we consider the flat integer lattice that

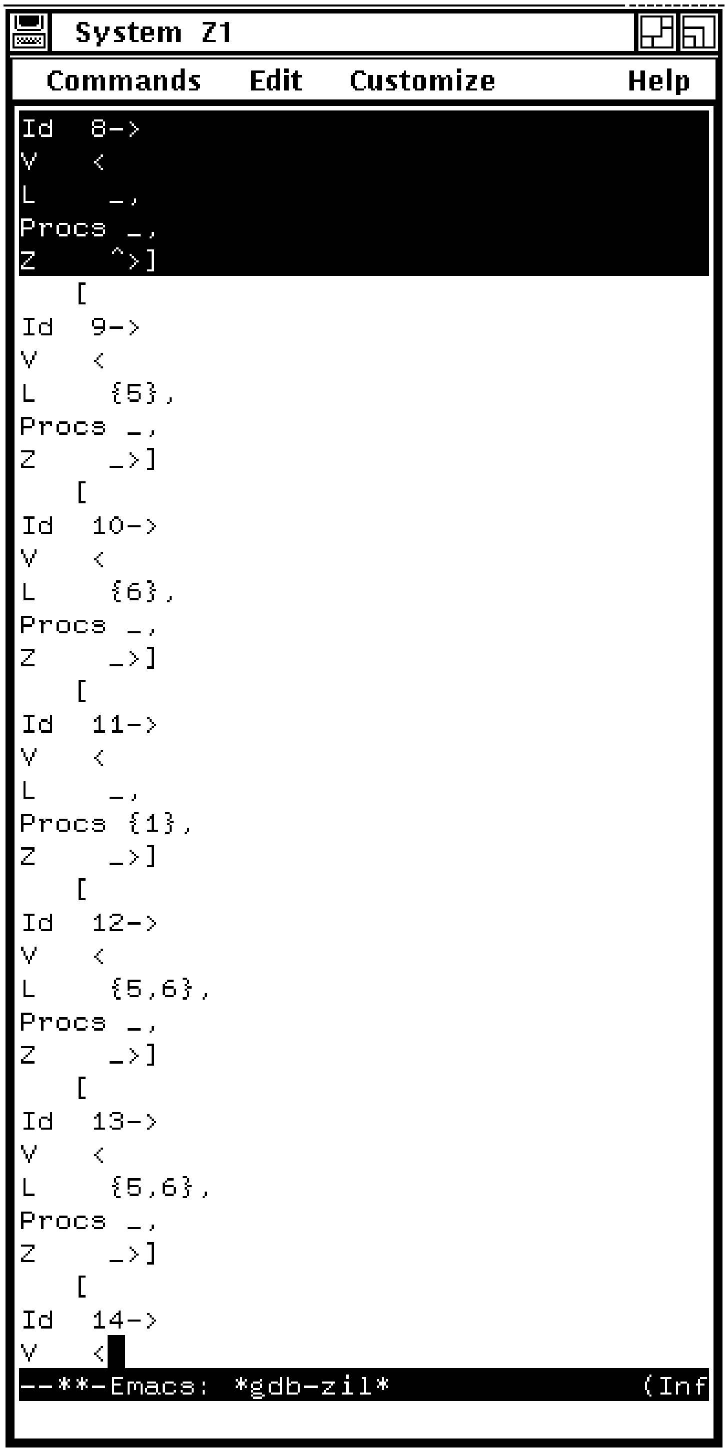


Figure 5.16: Integer Lattice Affects Alias Accuracy (part 1)

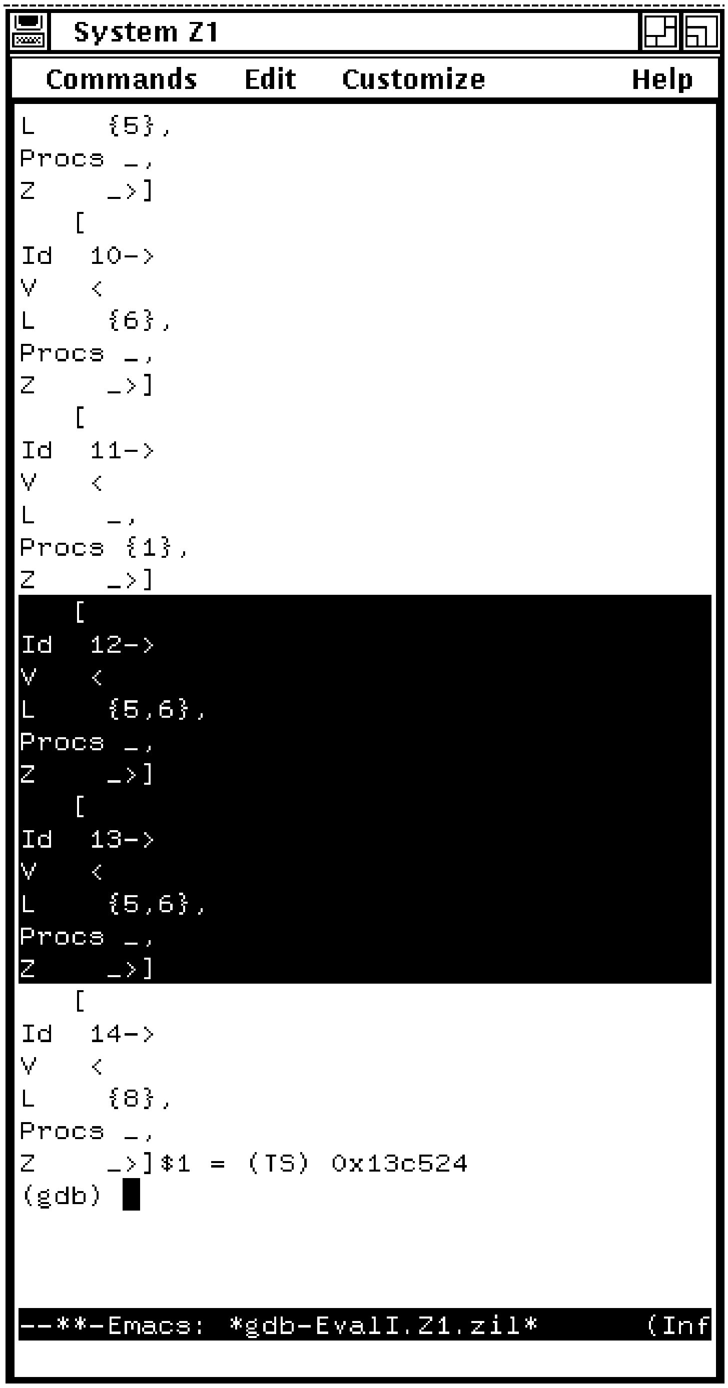


Figure 5.17: Integer Lattice Affects Alias Accuracy (part 2)

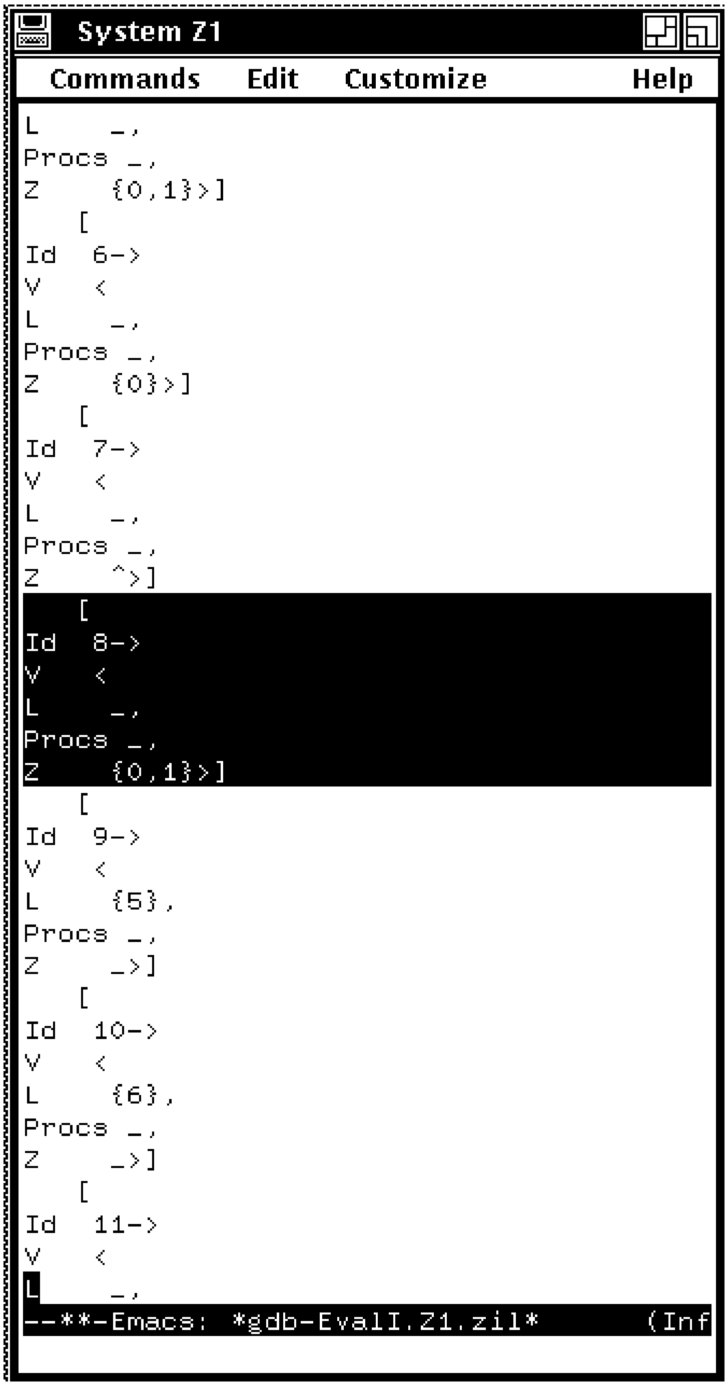


Figure 5.18: Integer Lattice Affects Alias Accuracy (part 3)

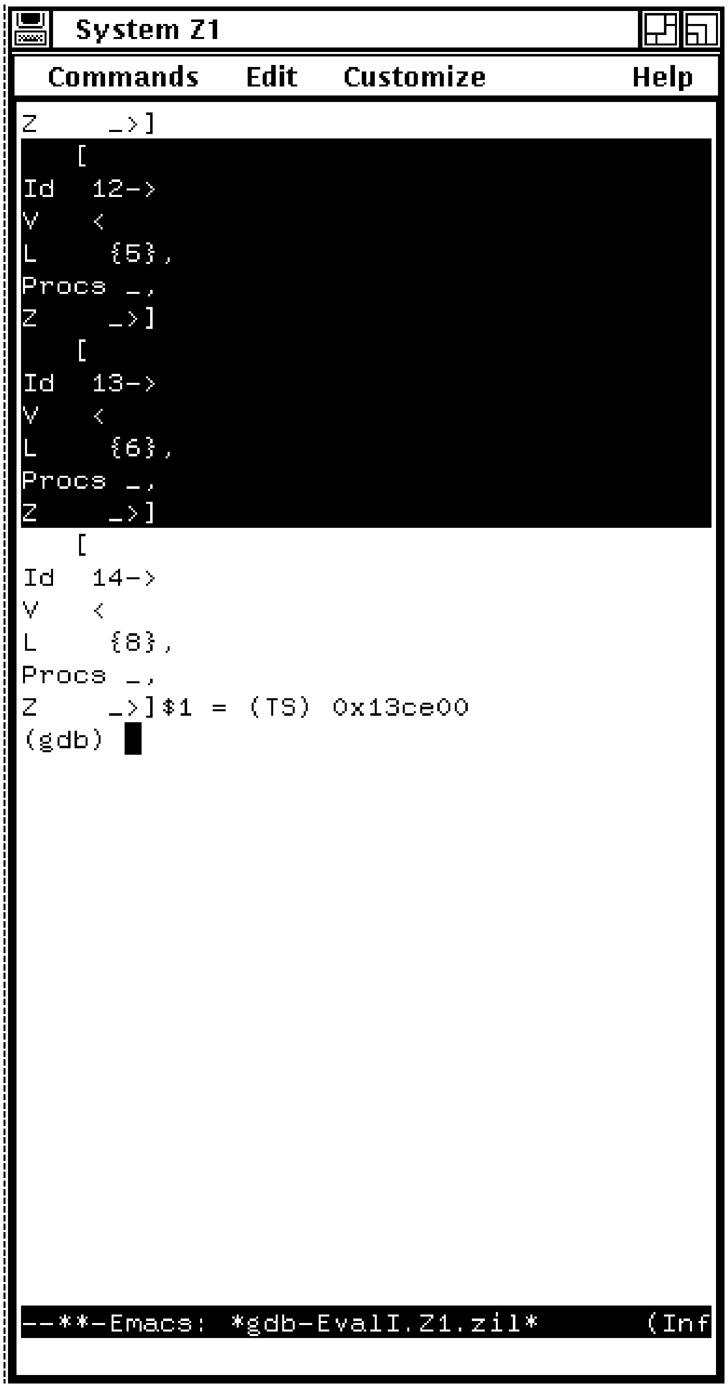


Figure 5.19: Integer Lattice Affects Alias Accuracy (part 4)

```
void f(int x, int *y, int *z) {
  if (x < y) h(*y, *z);                    (1)
  else h(*z-10,*y+10);                     (2)
}

void h(int l, int m) {                     (3)
  int i, sum = 0;
  for (i=0; i<l+m; i++) sum++;
}

void main() {
  int i, j, k;
  j = 10;
  k = 20;
  for (i=0; i<100; i++)
    f(i, &j, &k);                          (4)
}
```

Figure 5.20: C program `cp.c`

results from `(project Z (height> 1))`, and the flat location lattice that is given by `(project L (height> 1))`.

Consider the C program in Figure 5.20. At line 4, procedure `f` is called iteratively with the index value `i` and the locations of `j` and `k`. Inside `f`, another procedure `h` is called in both branches of the if expression. The formal parameters `l` and `m` of procedure `h` are initialized with the values of the locations pointed to by `y` and `z`, respectively (line 1), or with `*z-10` and `*y+10`, respectively (line 2). Note that, in any case, procedure `h`'s arguments values are constants (10 for `l` and 20 for `m`). Both calls to procedure `h` with expressions `h(*y,*z)` and `h(*z-10, *y+10)` pass the same values to its parameters.

Our analysis detects this. Firstly, the bound values of procedure `f`'s parameters `x` (Id 14)), `y` (Id 15), and `z` (Id 16) are shown in Figure 5.21. The top integer is recorded for the first parameter `x` (Id 14), and a constant location (Id 6 for `i` and Id 7 for `j`) is recorded for the second parameter (Id 15) and the third one (Id 16). The if condition (`x < *y`) at line 1 evaluates to the top boolean value (because `x` has the top integer). Therefore both branches are evaluated and the results are joined. In each branch, the values of the formal parameters `l` and `m` of `h` are constant integers. The values of `l` (Id 9) and `m` (Id 10) are shown in Figure 5.22.

• Analysis of `cp2.c` (Figure 5.23) with Projections `(project Z (height> 1))` and `(project L (height> 1))`

We shall show that, depending on the location lattice structure, the accuracy of constant propagation varies. We will vary the projections for the location lattice and see how they affect the constant propagation analysis. Firstly, we consider a projection that makes the

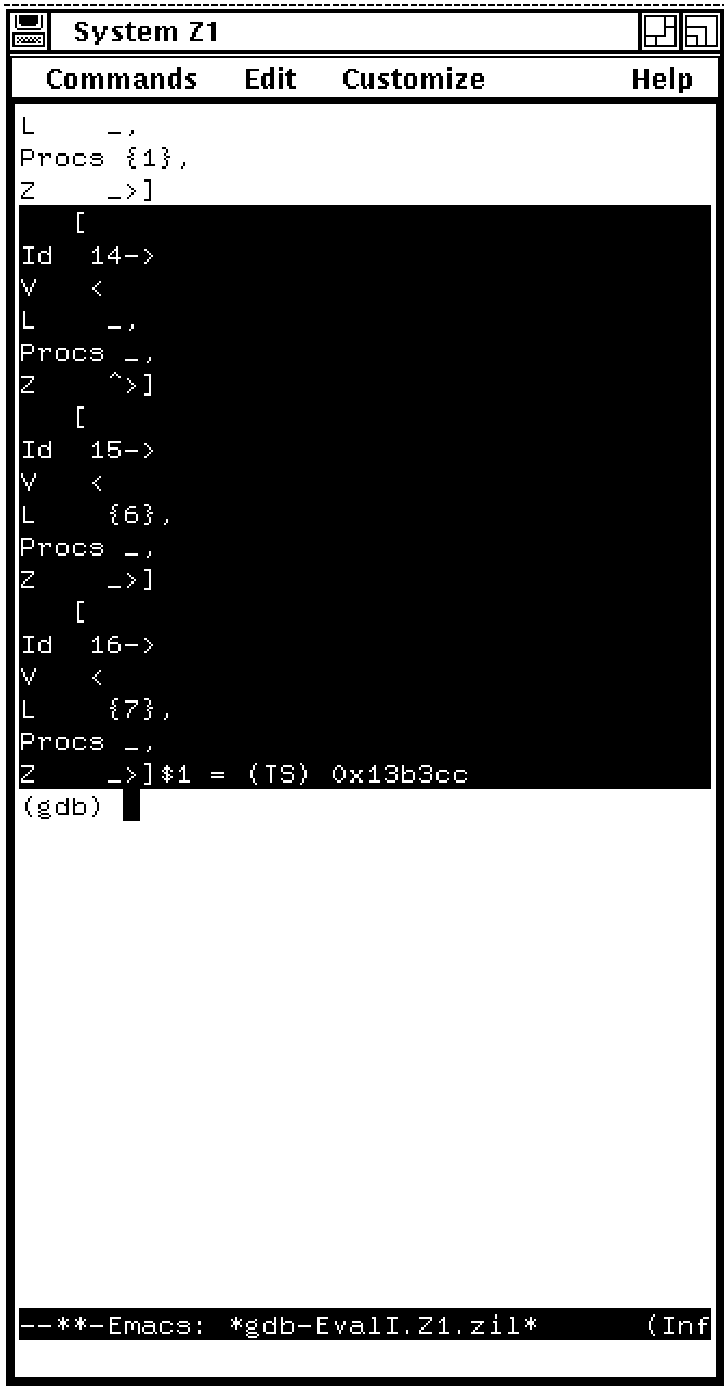


Figure 5.21: Analysis Result for `cp.c` (part 1)

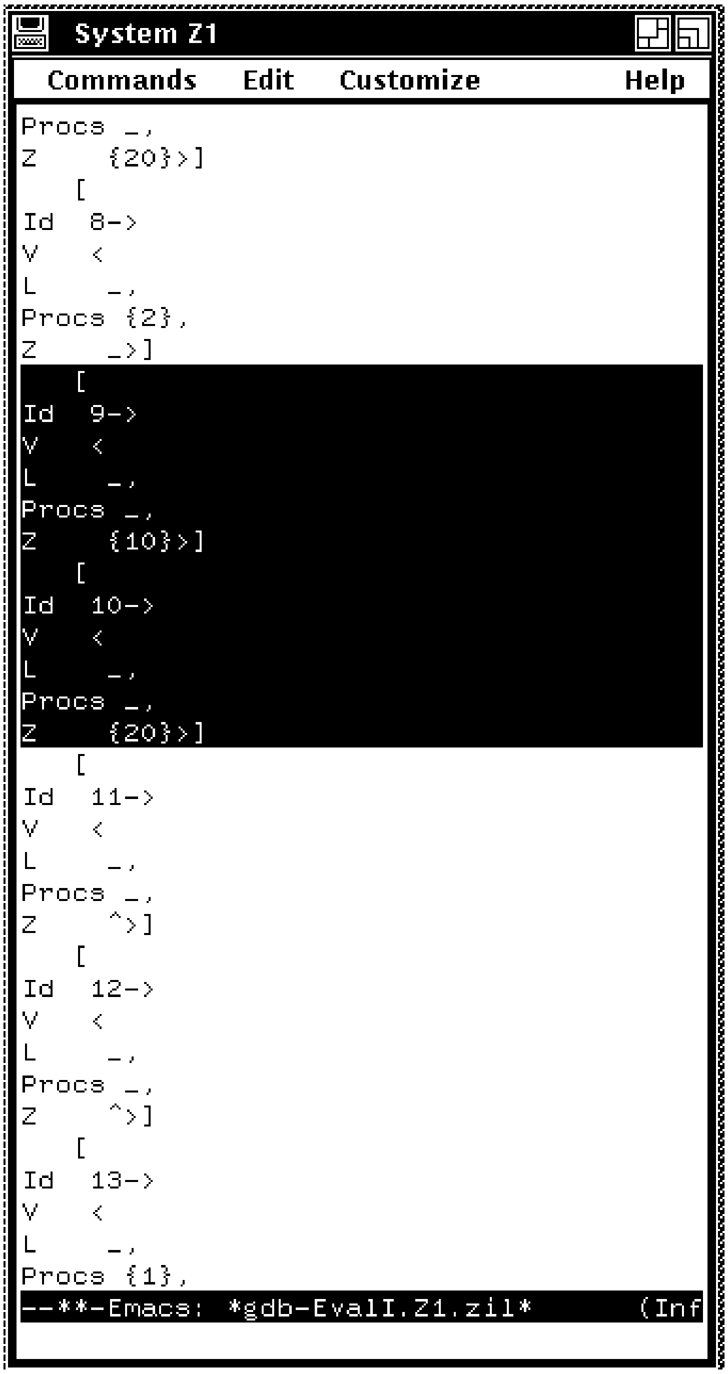


Figure 5.22: Analysis Result for `cp.c` (part 2)

```
/*
 * location lattice affects the cp accuracy: note the value of C
 */

void f(int *x, int *y) {
  if (*x < *y) (*x)++;              (1)
  else (*y)++;                      (2)
}

void main() {
  int i, j, k, C;
  C = 9;                            (3)
  j = k = 10;
  for (i=0; i<100; i++)
    if (i <= 50) f(&j, &k);         (4)
      else f(&k, &j);               (5)
}
```

Figure 5.23: C Program `cp2.c`

powerset location lattice into the flat one. Thus, when two distinct locations are evaluated for an expression, the best element we can use is the top location.

Consider the C program in Figure 5.23. Note that during the program execution variable `C` has a constant value. But our analysis result shows that `C` (Id 7) has the top integer (Figure 5.24).

This is because the increment expressions (`(*x)++` and `(*y)++`) inside `f` at line 1 and 2 access all the memory entries, because the location values bound to its formal parameters `x` and `y` are the top locations (Figure 5.25). This is, in turn, because the two calls to `f` in line 4 and 5 assign two distinct locations to each parameter. The flat location lattice forces the analysis to use the top location instead of two-element location values.

• Analysis of `cp2.c` (Figure 5.23) with Projection `(project Z (height> 1))` and No Projection of Locations

Now, when we do not apply any projection to the location lattice, the location values bound to the formal parameters `x` (Id 9) and `y` (Id 10) of procedure `f` are restricted to two-element location values. See Figure 5.26. This causes the analysis to leave the store entry for `C` intact, thus the value of `C` is determined to be a constant. See Figure 5.27.

5.4.2 Eval2

Eval2 is a more elaborate abstract interpreter than **Eval1**. We put additional things to **Eval1** in order that it's store has a notion of distinct instances of program variables. This is

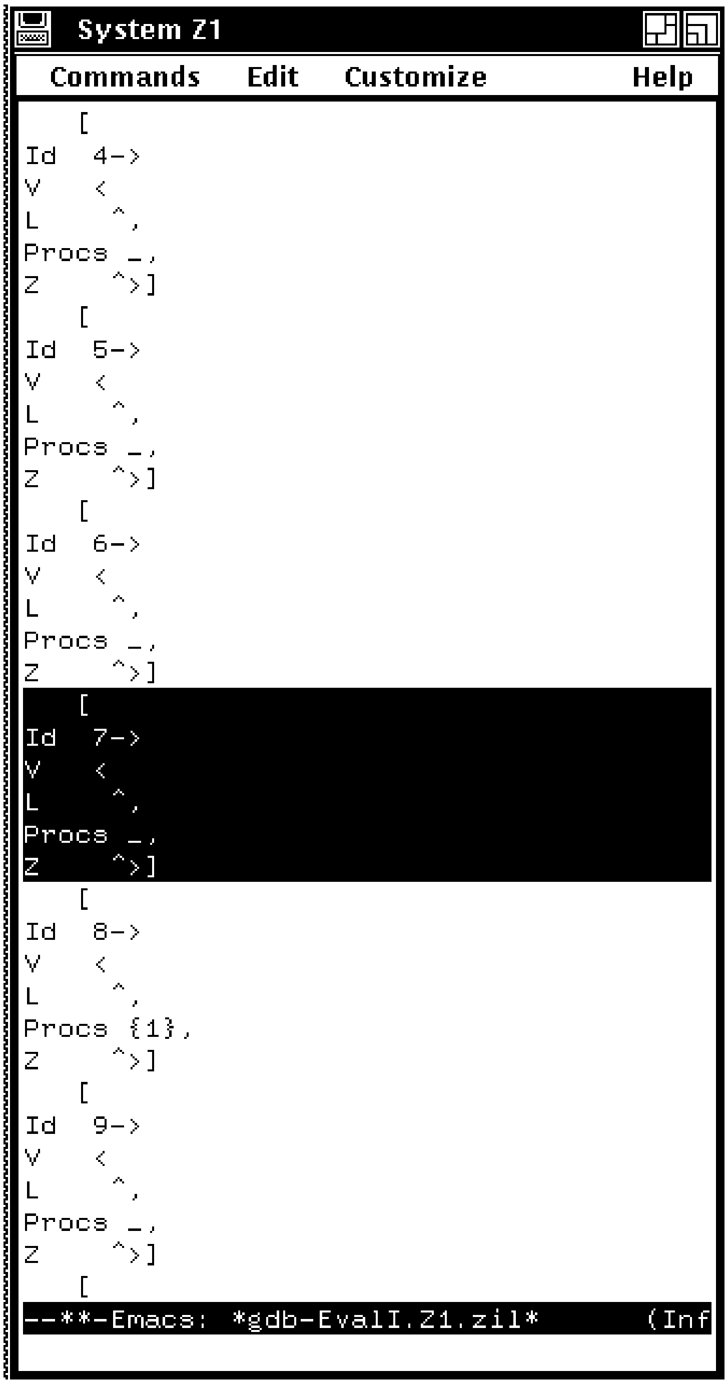


Figure 5.24: Analysis Result of `cp2.c` For `C` With Flat Location

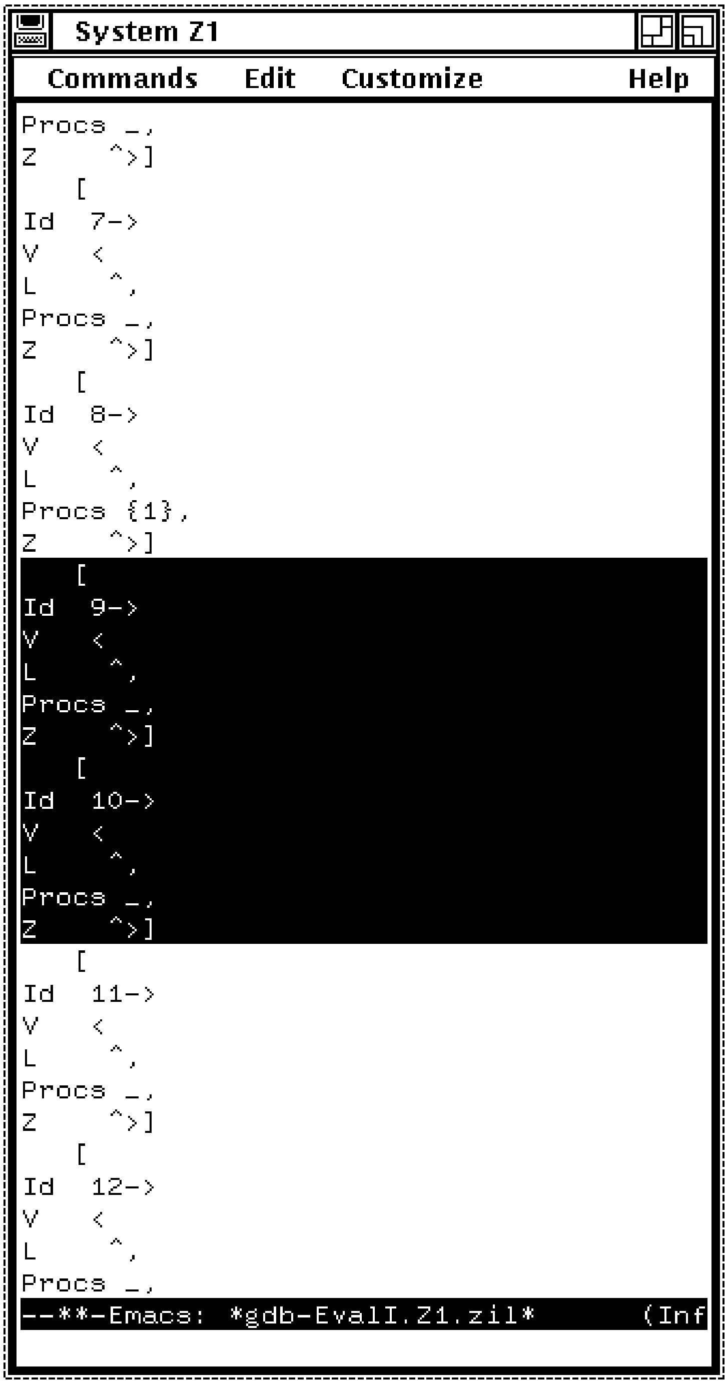


Figure 5.25: Analysis Result of `cp2.c` For `x` And `y` With Flat Location

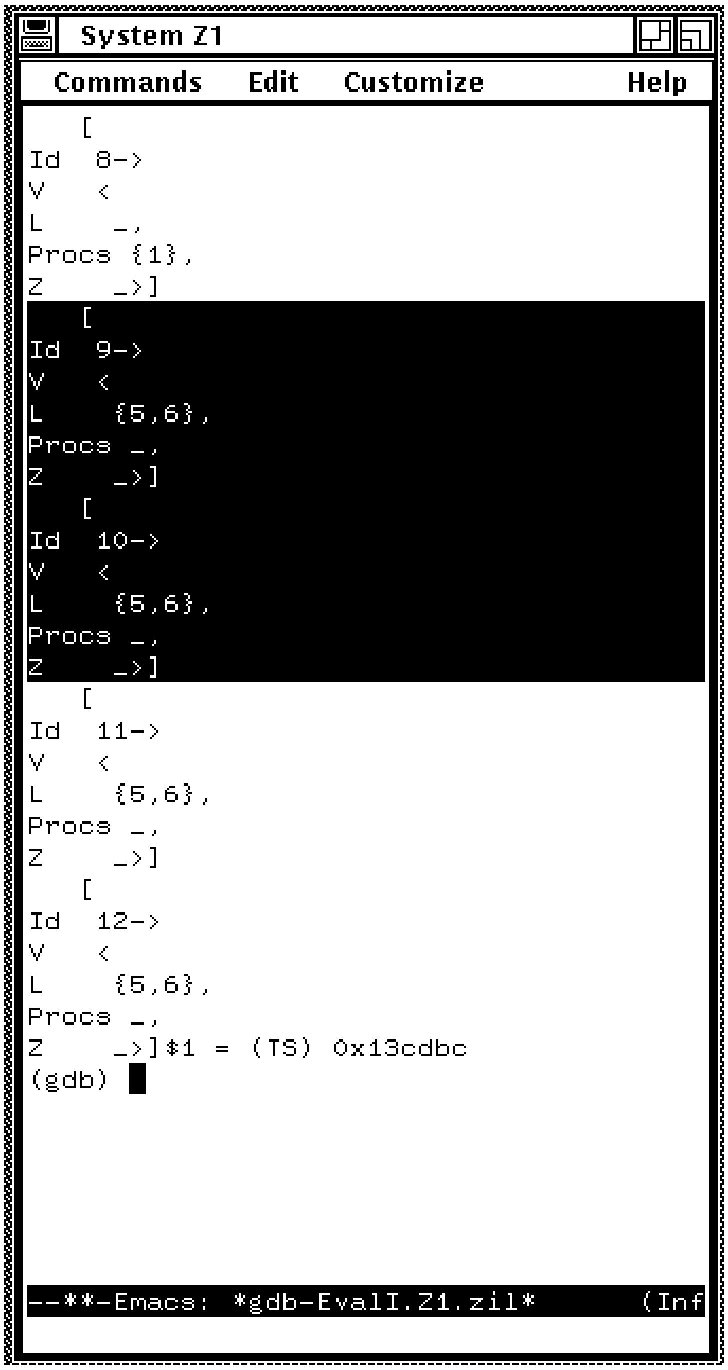


Figure 5.26: Analysis Result for `x` And `y` Without Projection For Location

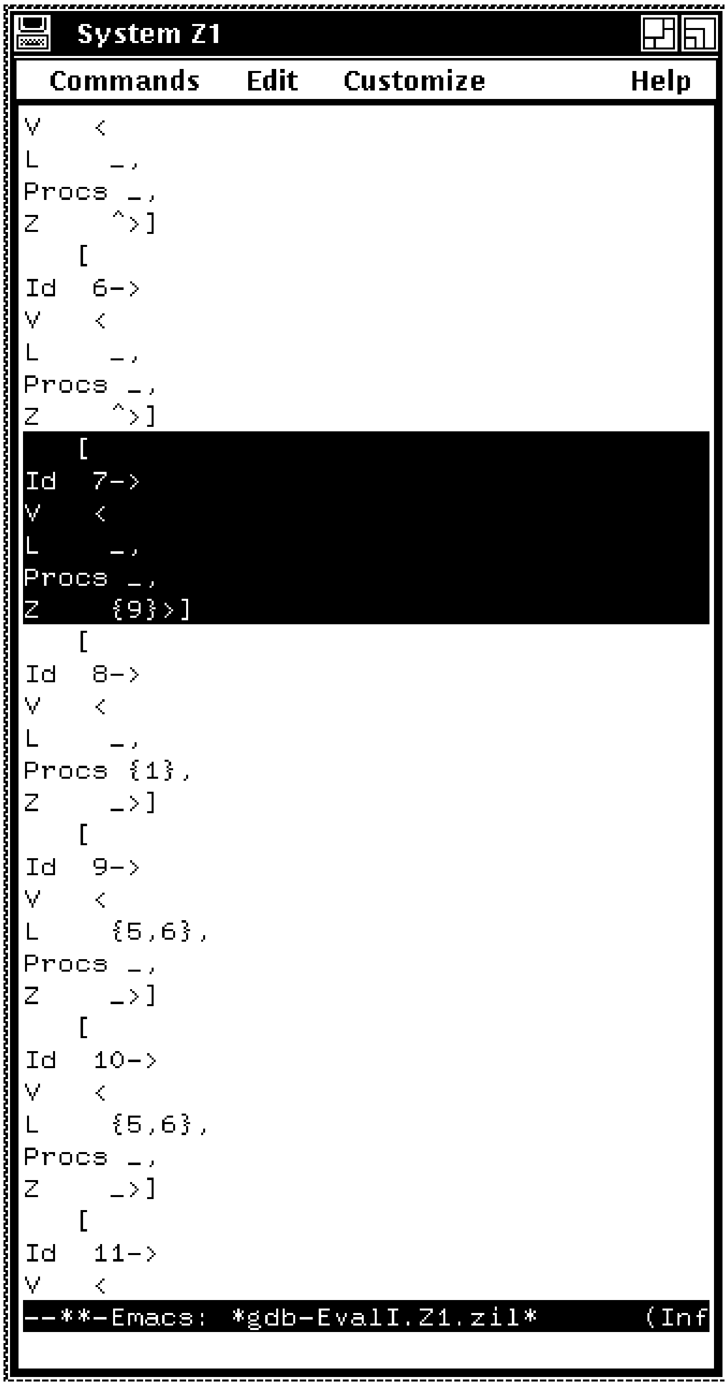


Figure 5.27: Analysis Result for C Without Projection for Location

desirable for the reason that when all instances of a variable are mapped to a single abstract instance, the abstract instance can never be overwritten by a write expression; rather, the previous conglomerate value is joined with the newly written value.

When several instances of a variable may exit simultaneously in the store, the abstract interpreter needs an environment that determines the current accessible instances for each variables. Also, when it evaluates a language construct that generates new instances of local variables (this is the case for procedure call in MIL), it needs something that differentiates these new instances from others. Thus, the pre-state `PRE` in **Eval2** is a tuple of environment `E`, store `S`, and procedure string[3] `P`.

```
(domain PRE (* E S P))
```

When new instances of local variables are allocated, the procedure string differentiates these new instances from others. A procedure string is a function that maps each procedure index into the number of activation records it has on the current stack. This infinite space (because, for a recursive procedure, an indefinite number of activation records can be stacked) is abstracted into three classes: `EPS` (epsilon), `D`, or `DD`. `EPS` indicates that a procedure is not active. `D` represents the case where only one activation record of a procedure is on the stack. `DD` represents the case where more than one activations of a procedure is pending (by recursive calls) on the stack during execution.

```
(domain P (-> Proc Gs))
(domain Gs (2^ setG))
(set setG (elements EPS D DD))
```

An instance of an identifier is determined by its owner procedure's stack configuration, as represented by the procedure string.

```
(domain E (synonym P))
```

The abstract store is now a two-level function: first it maps an identifier index to its image, and the image maps an instance to a value.

```
(domain S (-> Id SS))
(domain SS (-> G V))
(domain G (flat setG))
```

Note that the second-level store `SS` maps single instance specifier (in `G`) to its value, while the environment maps each identifier to multiple instance specifiers (in `Gs`). When we access a location whose instance gs has multiple elements of `setG`, we join the store entries `(@ ss g)` for each g in gs.

[3] We borrow this term from [Har89], where a much more elaborate version than ours is used.

```
(types
 (domain PRE (* E S P))              ; pre-state
 (domain POST (* S V))               ; post-stae
 (domain S (-> Id SS))               ; store
 (domain SS (-> G V))                ; second-level store
 (domain V (* L C Z))                ; value
 (domain L (* Ids P))                ; location
 (domain C (* Procs E))              ; closure
 (domain P (-> Proc Gs))             ; procedure string
 (domain E (synonym P))              ; environment
 (domain Ids (2^ setId))
 (domain Id (flat setId))
 (domain Procs (2^ setProc))
 (domain Gs (2^ setG))               ; instance specifiers
 (domain G (flat setG))              ; instance specifier
 (domain Z (2^ setZ))
 (domain T (flat setB))
 (domain Proc (flat setProc))
 (set setG (elements EPS D DD))
 (set setId (index NumOfPlaces))
 (set setProc (index NumOfProcs))
 (set setB (elements True False))
 (set setZ (range -999999 999999))   ; finite range of integers
)
```

Figure 5.28: Abstract Domains for **Eval2**

An abstract location is a pair of variable indices and an environment. The associated environment determines each variable's instance.

```
(domain L (* Ids E))
(domain Ids (2^ setId))
(set setId (index NumOfPlaces))
```

An abstract closure `C` is a pair of procedure indices and an environment.

```
(domain C (* Procs E))
(domain Procs (2^ setProc))
(set setProc (index NumOfProcs))
```

The abstract domain definitions are summarized in Figure 5.28.

Auxiliary functions used in **Eval1** are carried over to **Eval2** without changes. Some additional auxiliary functions are defined.

- `(instanceP (from setId P) (to Gs))` returns the instance of a variable, which is recorded in the procedure string that is passed as the second argument.
- `(circleS (from S setId Gs) (to V))`

  `(circleS s id gs)` returns the value in `s` of the variable `id` whose instance is `gs`.

- (moveP (from P setProc) (to P)), (moveGs (from Gs) (to Gs)) and (moveG (from setG) (to (

  The current stack configuration p $\in$ P is changed when a procedure pid is called: (moveP p pid).

```
(moveP (from P setProc) (to P))
(mfunction moveP (p sproc)
          (begin
           (let proc (Proc sproc))
           ([/] p (by proc
                    (moveGs (@ p proc)))
               )))

(moveGs (from Gs) (to Gs))
(mfunction moveGs (gs)
          (mapjoin (lambda (from setG) (to Gs) (g)
                   (moveG g))
                 gs))

(tfunction moveG (from setG) (to Gs)
          (EPS  (Gs D))
          (D    (Gs DD))
          (DD   (Gs DD))
          )
```

- (slashSS (from SS V Gs) (to SS))

  (slashSS ss v gs) updates the store entry ss with a new value v for the instances in gs.

```
(mfunction slashSS (ss v gs)
          (join ss
               (map[/] ss
                      (lambda (from setG) (to V) (g) v)
                      gs)))
```

- (OverShadow (from E P Ids Ids) (to E))

  (OverShadow e p ids ids') returns a new environment whose entries for the identifiers in ids and ids' are overshadowed by the correspondents in p.

```
(mfunction OverShadow (e p params locals)
          (begin
           (let ids :Ids (join params locals))
           (let owners :Procs (join (for i 1 (|| ids)
                                    (Procs
                                      (Owner (nth ids i))))
                                ))
           (map[/] e
                  (lambda (from setProc) (to Gs) (f)
                    (@ p (Proc f)))
                  owners))
           )
```

The evaluation body of **Eval2** is largely identical with that of **Eval1** except for `read`, `write`, `id`, `closure`, `create`, and `call` expressions.

- For a `read` expression, the location value contains the instance component in addition to a set of variable indices. For each instance of each variable, it access the store and joins the stored values.

```
(isRead x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 S))
 (let v1 (! post1 V))
 (let l (! v1 L))
 (let ids (! l Ids))
 (let u (! l P))            ; instance specifier
 (let v2 (mapjoin (lambda (from setId) (to V) (i)
                   (circleS s1 i (instanceP i u)))
                 ids))
 (POST s1 v2)
 )
```

- Similarly, for a `write` expression, we update the location with new values, but we must handle a location that contains the instance component.

```
(isWrite x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let v1 (! post1 V))
 (let post2 (Eval (NthSubexpr x 2) e (! post1 S) p))
 (let s2 (! post2 S))
 (let v2 (! post2 V))
 (let l (! v1 L))
 (let ids (! l Ids))
 (let u (! l P))             ; instance of the location to write
 (let s3 (mapjoin
         (lambda (from setId) (to S) (id)
           ([/] s2
                (by (Id id)
                    (slashSS (@ s2 (Id id)) v2 (instanceP id u)))))
         ids))
 (POST s3 v2)
 )
```

- For a `create` expression, the newly generated location contains an instance, which is determined by the procedure string of the pre-state.

```
(isCreate x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1  (! post1 1))
 (let id  (DynamicId x))
 (let ss  (@ s1 (Id id)))
 (let l (L (Ids id)
```

```
          ([/] (P bottom) (by (Proc (Owner id))
                              (instanceP id e)))))
 (let v (V l (C bottom) (Z bottom)))
 (POST s1 v)
 )
```

- For a `closure` expression, the current environment is packaged in the closure value in addition to the procedure index.

```
(or (isProc x) (isClosure x))
(POST s (V (L bottom)
          (C (Procs (ProcId x)) e)
          (Z bottom)))
```

- For a `variable` expression, its location's instance component is determined by the current environment.

```
(isId x)
(begin
 (let id (Xid x))
 (let l (L (Ids id)
          ([/] (P bottom)
               (by (Proc (Owner id))
                   (instanceP id e)))))
 (POST s (V l (C bottom) (Z bottom)))
 )
```

- For a `call` expression, before we evaluate the procedure body with the environment at the definition of the procedure, the procedure string (of the closure being invoked) should be used for the instances of the newly allocated local variables. For this reason, the environment needs to be overshadowed by the new procedure string entries for the procedure's parameters and local variables. Otherwise, this is similar to the corresponding case in **Eval1**.

```
(or (isCall x) (isApply x))
(begin
 (let n (NumOfSub x))
 (let num_actual (- n 1))
 (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 s1 (! (_post n) S))
 ;
 ; get the closure value
 ;
 (let c (! (_v 1) C))
 (let procs (! c 1))    ; closure's procedure indices
 (let f (! c 2))        ; closure's environment
```

```
;
; for each procedure index and environment of the closure value,
; bind parameters and evaluate its body with the environment
;
(mapjoin
 (lambda (from setProc) (to POST) (alpha)
    (begin
    (let mom (Body alpha))
    (let env1 (moveP f alpha))
    (let p1 (moveP p alpha))    ; new procedure string because of this call
    (let num_param (NumOfParam alpha))
    (let num_local (NumOfLocal alpha))
    (if (< num_actual  num_param)
        (POST bottom)
      (begin
       (let params :Ids (join (for i 1 num_param
                                (Ids (NthParam alpha i)))))
       (let locals :Ids (join (for i 1 num_local
                                (Ids (NthLocal alpha i)))))
       ;
       ; overshadow env1 with the entries of p1 for the locals and params
       ;
       (let env2 (OverShadow env1 p1 params locals))
       ;
       ; parameter binding
       ;
       (let (_s2 0) s1
            (_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))
                                           (@ env2 (Proc alpha))))
                          ([/] (_s2 (- i 1)) (by (Id param) ss)))))
       (Eval mom env2 (_s2 num_param) p1)
       ))))
   procs))
```

• Analysis of `alias.c` (Figure 5.29) with Projection `(project Z (height> 1))` and No Projection of Locations

Consider the C program that we used before. We repeat it in Figure 5.29.

The first instance of `y` by the initiating call to `f` in line 5 contains the location of `j`, while its other instances by the recursive calls to `f` in line 2 contain the location of `i`. In our new abstract interpreter we use two distinct tokens **d** and **dd+** to differentiate these two class of instances. We differentiate the variable instances between the one allocated by the initiating call to its owner procedure and those allocated by subsequent recursive calls.

The first formal parameter `x` (Id 8) of `f` has the location of `i` (id 4) as the value of each of its instance. The last store image for `x` (Figure 5.30) shows this: note the entries under `D` and `DD`. Now, the second formal parameter `y` (Id 9) contains the location of `i` (Id 4) or `j` (Id 5), but different locations are found in distinct instances of `y`. Its instance for the initiating

```
#define UNKNOWN (-0x8000000)

void f(int *x, int *y) {
  int m;
  if (UNKNOWN) {
    m = *x;
    f(x,x);
  }
  else m = *y;
}

void main () {
  int i, j;
  int *a, *b;

  i = j = 0;
  a = &i;
  b = &j;
  f(a,b);
}
```

Figure 5.29: C Program `alias.c`

call to `f` at line 5 contains the location of `j` (Id 5) whereas the instances from subsequent recursive calls to `f` at line 1 contain the location of `i` (Id 4). The last store image for `y` is shown in Figure 5.31.

5.4.3 Eval3

Eval2 introduced abstract instances that differentiate distinct instances of program variables. It uses two instance tokens, one for the instances allocated by non-recursive calls and one for those allocated by recursive calls. In **Eval3** we will use more elaborate store which, in addition to the distinct locations for different instances of program variables, will have distinct locations for separate array elements and structure fields. Note that in **Eval1** or in **Eval2**, distinct elements of an array variable or distinct fields of a structure variable have a single location in the abstract store.

The pre-state `PRE` is the same as for **Eval2**. The store is now a three-level function: first it maps an identifier index to its image, the image maps an instance to the final image, which maps a pair of an offset and a size to a value.

`(domain S (-> Id SS))`	from id
`(domain SS (-> G B))`	from an instance to a block
`(domain B (-> I V))`	from a pair of offset/size to a value

The block `B` models the store region for array or structure variables. It represents a block of contiguous values in memory. Array elements or structure fields are accessed by providing

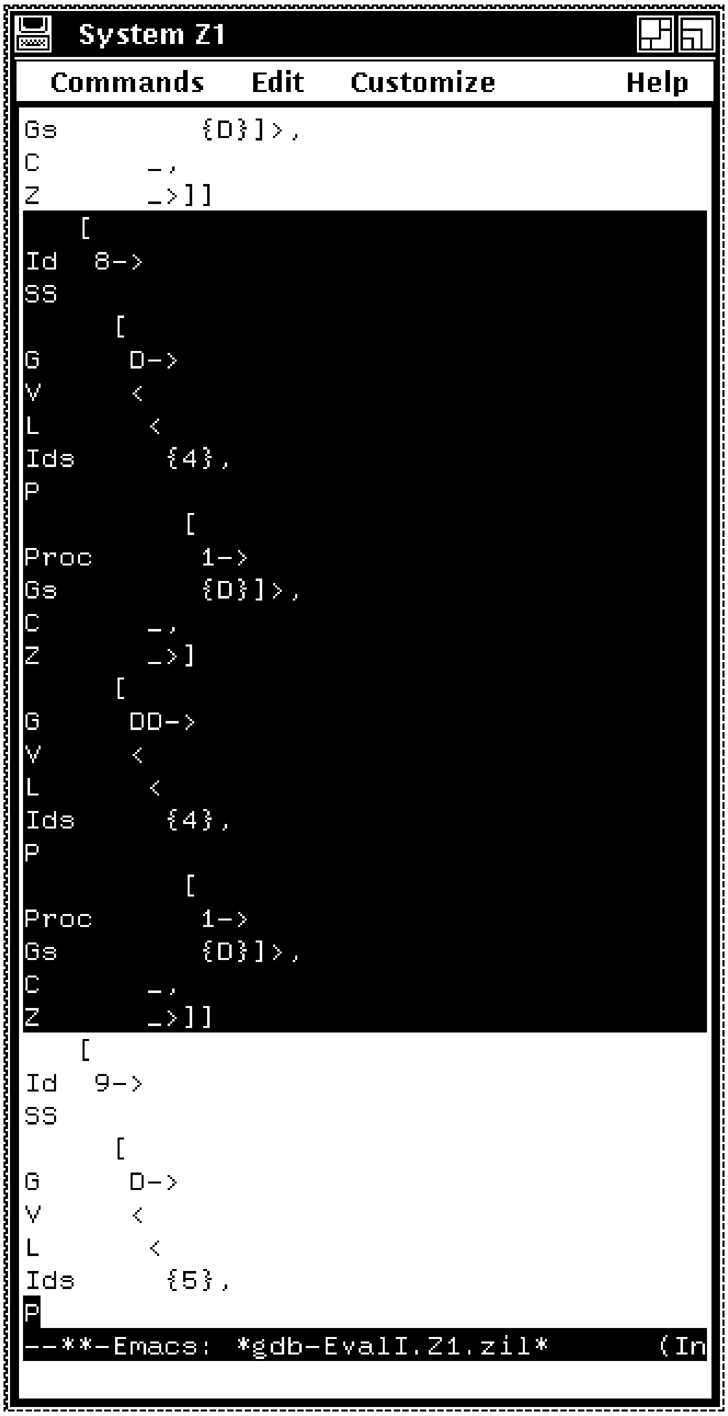


Figure 5.30: Last Store Image of `alias.c` For `x` From **Eval2**

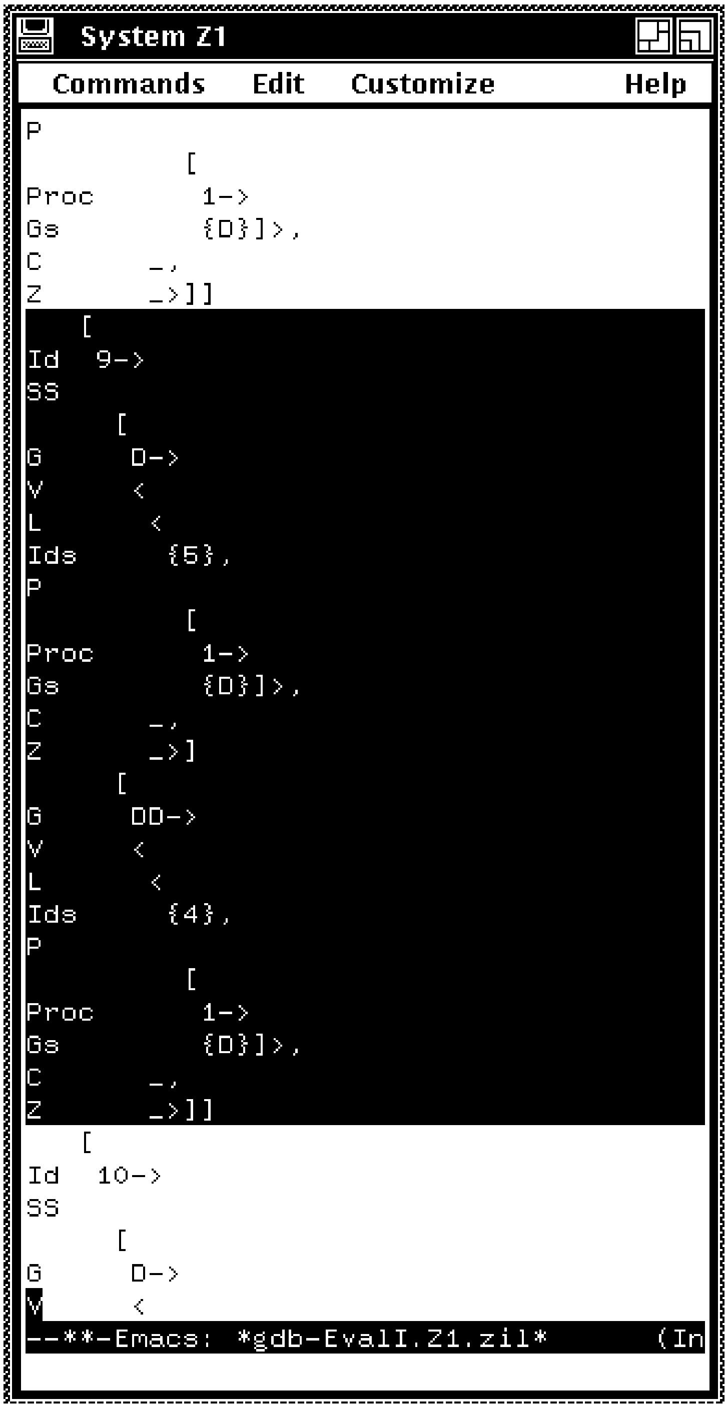


Figure 5.31: Last Store Image of `alias.c` For `y` From **Eval2**

a pair `I` of an offset and a size:

```
(domain I (flat setI))
(set setI (* setO setY))
```

The offset `setO` is a range set

```
(set setO (range 0 40))
```

When offset is larger than the upper-bound, it causes the top offset/size pair to be used. The size `setY` is the number of bytes for each type of value (e.g., 4 for integer, 8 for long, 8 for closure, etc.)

```
(set setY (range 1 16))
```

New auxiliary functions and evaluation rules are included in Appendix D. The `read`, `write`, and `call` expression requires some complicated manipulations of the block index values. The reader is recommended to see [Har92] or in [YH92] for those operations written in conventional mathematical form. Ours are just a transliteration of those definitions into Z1.

• Analysis of `array.c` (Figure 5.32) with Projection `(project (height> 1))` and No Projection of Locations

Consider the C program `array.c` in Figure 5.32. An array `a` and a structure variable `xy` are passed into a function and are assigned distinct values at distinct array elements and structure fields. When we analyze the program with **Eval1**, which uses a single location for distinct elements of an array variable, the two assignments (at line 5 and 6) to two distinct fields of the structure variable `xy` causes the store entry for `xy` to have the top integer. This is because in **Eval1**, distinct fields of a structure variable share a single slot in the abstract store. Assigning distinct values for this single store entry deposits the top value in that entry. Similarly, the assignments (line 1 to 4) to distinct array elements (indexed by the distinct fields values of `xy`) causes the top integer to be deposited in the array variable. Figure 5.33 shows this result. The array `grid` has index 4 and the structure variable `xy` has index 5. The store images of both variables contain the top integer.

Now, compare this with the result from **Eval3**. The last store image for the structure `xy` (Id 5) is in Figure 5.34. Its distinct fields have distinct values, as assigned in line 5 and 6.

The procedure `f`, which receives the addresses of the array `a` and structure `xy`, assigns distinct values to distinct elements of `a`, using the two fields of `xy` as indices. The two `if` branches in fact assign the same value to each array element. The analysis result (last store image) for the array `a` (Id 6) shows this information; See Figure 5.35. The second element of `a` (the entry under `<4,4>` (offset 4, size 4)) has integer 10, and the sixth element (the entry under `<20, 4>` (offset 20, size 4)) has 11.

```
struct XY {
  int x;
  int y;
};

#define UNKNOWN (-0x8000000)

void f(int grid[], struct XY *place)
{
  if (UNKNOWN) {
    grid[place->x] = 10;                       (1)
    grid[place->y] = grid[place->x] + 1;       (2)
  }
  else {
    grid[place->y] = 11;                       (3)
    grid[place->x] = grid[place->y] - 1;       (4)
  }
}

#define SIZE 10

void main()
{
  int a[SIZE];
  struct XY xy;
  int i;

  xy.x = 1;                       (5)
  xy.y = 5;                       (6)
  f(a, &xy);                      (7)
}
```

Figure 5.32: C program `array.c`

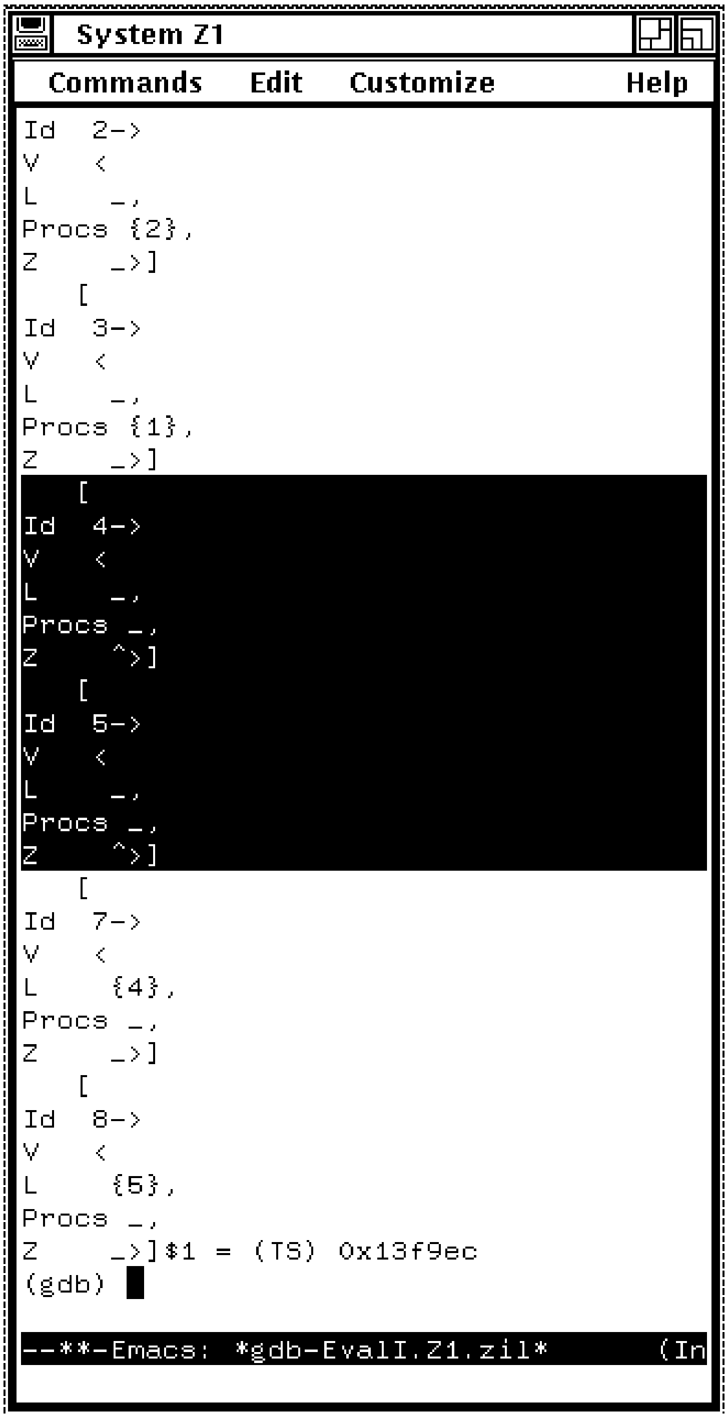


Figure 5.33: Last Store Image For Array `a` And Structure `xy` From **Eval1**

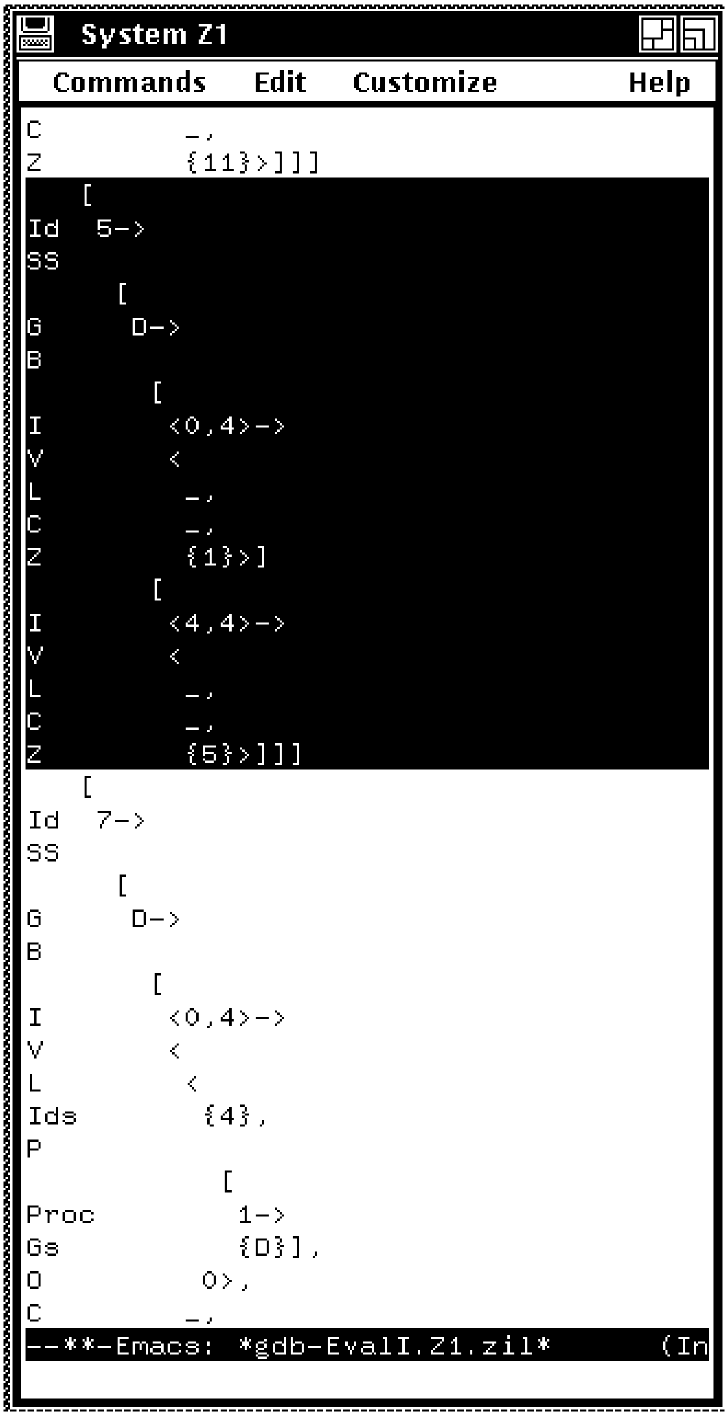


Figure 5.34: Last Store Image For Structure **xy** From **Eval3**

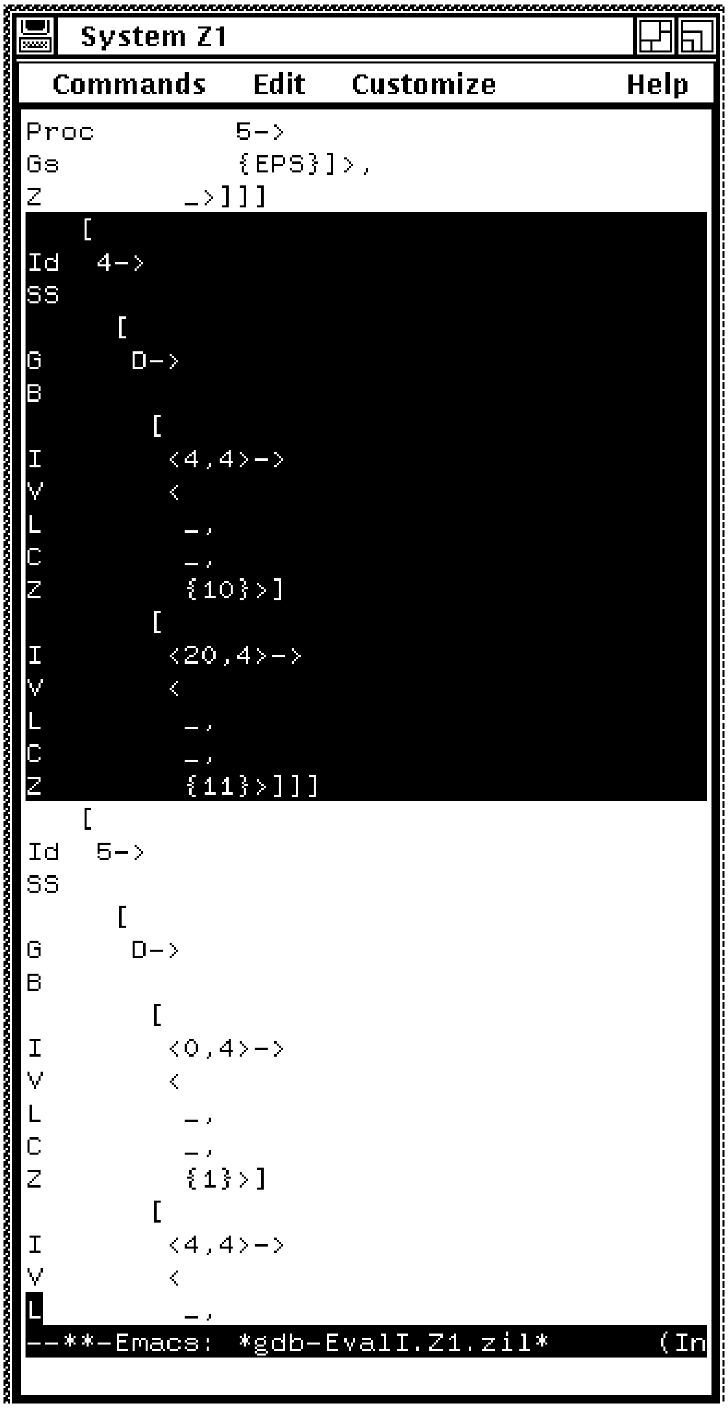


Figure 5.35: Last Store Image For Array a From **Eval3**

5.5 Def-use/Use-def Chain Analysis

The abstract interpreter for def-use/use-def chain analysis is an instrumentation of **Eval1**, which we used for constant propagation and alias analysis. Our def-use/use-def chain analysis will record which expression defines a value in each memory location. Our def-use/use-def chain analysis could have been based upon **Eval3**, which uses the most elaborate store abstraction. That we avoided here because it would complicate our presentation. The reader however can see that **Eval3** can be instrumented using the same idea without any difficulty.

The store entry for each location (in this case, an identifier index) has a pair `W` of its value `V` (abstract integer, location, or closure) and its definition point `X`.

```
(domain S (-> Id W))
(domain W (* V X))
(domain X (2^ setX))
(set setX (index NumOfExprs))
```

When a location is written with a value, it contains not only its new value but also the current expression index. There are two expressions in MIL that defines a memory location. One is the `write` expression and the other is the `call` expression (where parameters are defined).

Caveat: Since the abstract store maps all instances of a variable into a single abstract instance, an abstract location's value and its definition point can never be overwritten; rather, the previous conglomerate value (respectively, its definition point) is joined with the newly written value (respectively, with the current definition point).

- For a `write` expression, the location into which a value is written contains the current `write` expression index as its definition point. Note that the definition point is joined with the previous conglomerate value. This is because all instances of a variable are mapped to a single abstract instance.

```
(isWrite x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let loc (! (! post1 V) L))
 (let post2 (Eval (NthSubexpr x 2) (! post1 S)))
 (let s2 (! post2 S))
 (let v2 (! post2 V))
 (let def_point (X (setX (Xindex x))))   ; current expression point
 (let s3 (mapjoin
          (lambda (from setId) (to S) (id)
            ([/] s2
                 (by (Id id)
                     (W (join (! (@ s2 (Id id)) V)
                              v2)
                        ;
                        ; the location to write has this definition point
                        ;
```

```
                      (join (! (@ s2 (Id id)) X)
                            def_point))
                   )))
          loc))
 (POST s3 v2)
 )
```

- For a `call` expression, the locations for formal parameters are defined with actual argument values. Their definition points are recorded with this call expression. Note that the definition point is joined with the previous conglomerate value. This is because all instances of a variable are mapped to a single abstract instance.

```
...
;
; parameter binding
;
(let (_s2 0) s1
     (_s2 i) (for i 1 num_param
                  (begin
                   (let param :setId (nth params i))
                   ;
                   ; param's def point is this call expression
                   ;
                   (let def_point (X (setX (Xindex x))))
                   ([/] (_ s2 (- i 1))
                        (by (Id param)
                            (W (join (! (@ (_ s2 (- i 1))
                                           (Id param)) V)
                                     (_ v (+ i 1)))
                               (join (! (@ (_ s2 (- i 1))
                                           (Id param)) X)
                                     def_point))))
                   )))
 ...
```

After the analysis, the post-state image (a tuple of the store and the expression value) associated with each program point will provide us with the information necessary to derive the def-use/use-def chain. For each use point (`(read E)`), the location read (value of `E`) and the store image of the post-state of `E` shows which expression most recently defined the location that will be referenced by this `read` expression. From this use-def information, def-use information can easily be derived.

• Analysis of `du.c` (Figure 5.36) with Projection `(project Z (height> 1))` and No Projection of Locations

Consider the C program in Figure 5.36. The main procedure's local variables `i`, `j`, and `k` are defined inside procedure `f`. Statement 1, 2, or 4 inside `f` defines values of `i`, `j`, or `k`. Note that by the recursive call to `f` at line 3, the second and third parameters are swapped at the recursive call.

```
/*
 * For the uses of i, j, k in "S = i + j + k"
 * i is defined in three positions. j and k are defined in two positions.
 */

#define UNKNOWN (-0x80000000)

void f(int *x, int *y, int *z) {
  if (UNKNOWN) {
    *y = *x + 1;                (1)
    *z = *x - 1;                (2)
    f(x, z, y);                 (3)
  }
  else
    *x = 10;                    (4)
}

void main() {
  int i, j, k, S;
  for (i=0; i<100; i++)
    f(&i, &j, &k);              (5)
  S = i + j + k;                (6)
}
```

Figure 5.36: C program `du.c`

Let's examine the store image for `j` (Id 5) and `k` (Id 6) at the use point of line 6. See Figure 5.37. It shows that the value that is alive at the read point of line 6 is defined either at expression 32 or 40. Figure 5.38 shows that these two expressions correspond to statement 1 and 2 of the C program, respectively. This is correct information. Because of the recursive call to `f` at line 3, the two values passed to `y` and `z` are swapped repeatedly. Thus the assignment (line 1) of the location pointed to by `y` can define the location of `j` or `k`. Similarly, the assignment (line 2) of the location pointed to by `z` can define the location of `j` or `k`,a nd these defined values reach the use point at line 6.

• Analysis of `fptr.c` (Figure 5.39) with Projection `(project Z (height> 1))` and No Projection of Locations

Now consider the C program `fptr.c` in Figure 5.39 which passes function pointers across several procedures. Let's examine the overall structure of the program. Initially, `f` is called at line 8 with pointers to local variables `i` and `j` and with pointers to the two procedures `sum` and `multi`. Inside `f`, the passed procedures `op1` or `op2` are invoked with the location of either `i` or `j` (line 1 and 3). Inside the procedures `sum` or `multi`, which are passed as arguments to `f`, the variable `i` or `j` is defined (line 4 and 5). Note that the location of `j` is passed only at the initial call to `f` at line 8. Its subsequent recursive call at line 2 does not pass the location of `j`. Note also that at the recursive call, the procedure pointers bound

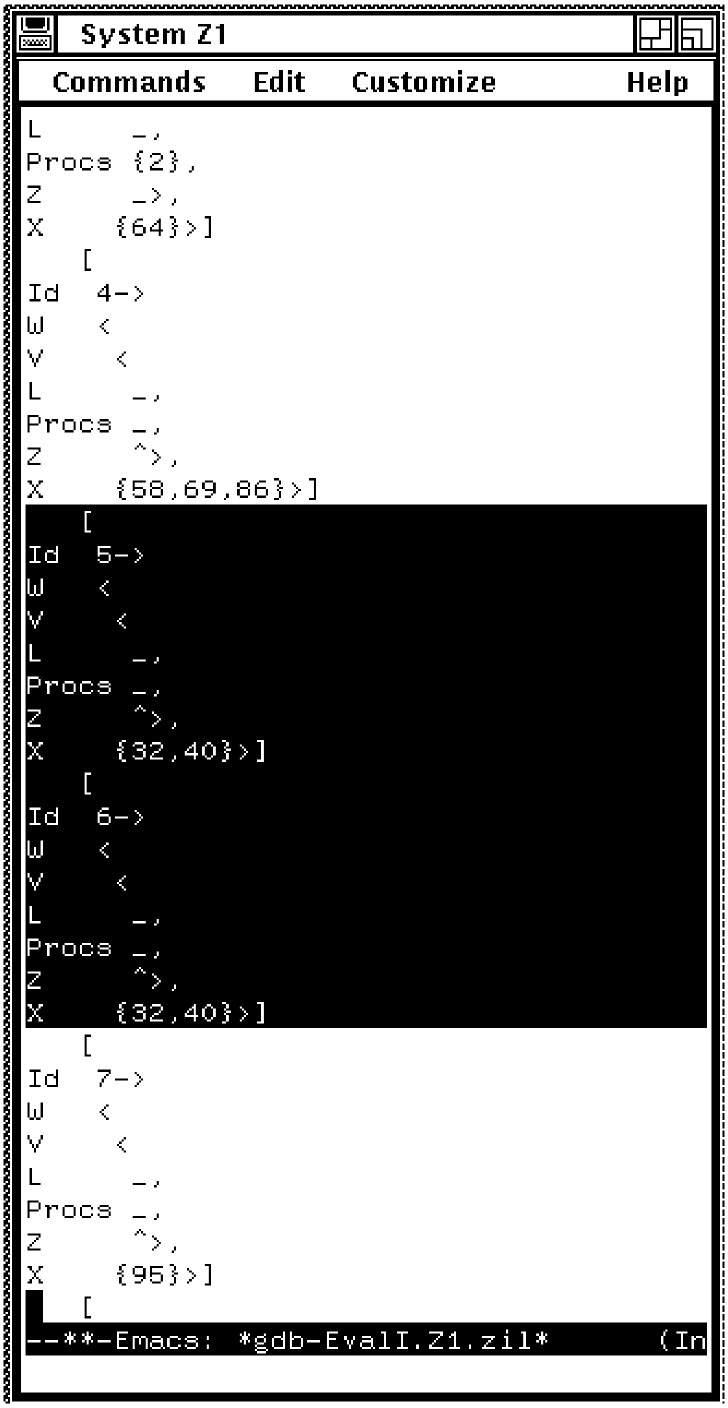


Figure 5.37: Store Image For `j` And `k` At Their Use Point (line 6 of `du.c`)

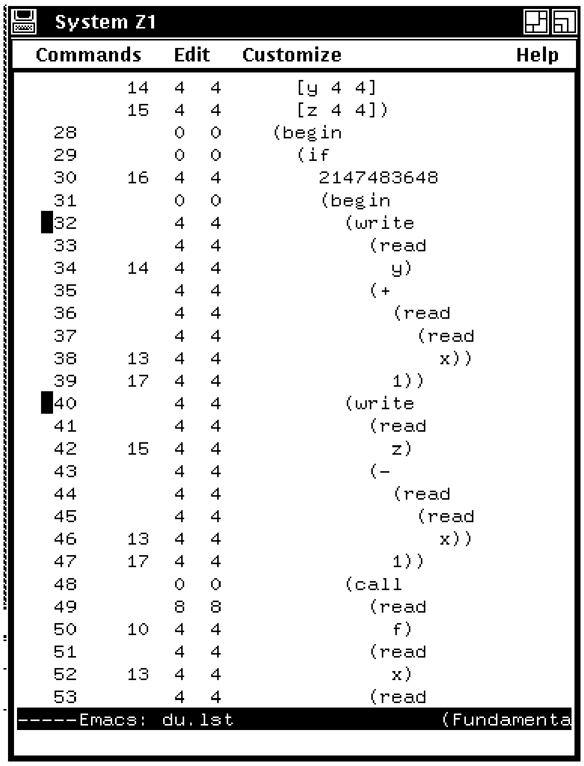


Figure 5.38: Expression Points 32 And 40 For MIL Version Of `du.c`

```
/*  fptr.c
 *  Note the values of op1 and op2, and the values of x and y.
 *  Note the definition points of i and j.
 */

#define UNKNOWN (-0x8000000)

void f(int *x, int *y, int op1(int*), int op2(int*)) {
  int m;
  if (UNKNOWN) {
    m = op1(x) - op2(y);                (1)
    f(x, x, op2, op1);                  (2)
  }
  else
    m = op1(x) + op2(y);                (3)
}

int sum(int *n) {
  int i, result;
  result = 0;
  for (i=0; i < *n; i++) result += i;
  *n = result;                          (4)
  return result;
}

int mult(int *n) {
  int i, result;
  result = 1;
  for (i=0; i<*n; i++) result *= i;
  *n = result;                          (5)
  return result;
}

void main () {
  int i, j, out;

  i = 10;                     (6)
  j = 20;                     (7)
  f(&i, &j, sum, mult);       (8)
  out = i + j;                (9)
}
```

Figure 5.39: C Program `fptr.c`

to the parameters `op1` and `op2` are swapped.

Consider the use points of `i` (Id 6) and `j` (Id 7) at the plus expression of the last statement of `main`. Let's examine their definition points. The store image for the variables at line 9 is in Figure 5.40. It shows that the value of `i` that is alive at this point is defined either at line 4 (expression 113), at line 5 (expression 160), or at line 6 (expression 174). This is correct since the location of `i` can be passed to both the `op1` and `op2` by the recursive call at line 2, thus it can be defined at line 4 or line 5. What about line 6 (expression 174)? Why did it compute that `i`'s value defined at line 6 may be reachable to line 9? This is because we can not kill the previous conglomerate definition points of a location, which is in turn because all instances of a variable is mapped to a single abstract instance. This is the reason why the definition point of `i` at line 6 is not overwritten by the definition points inside `sum` or `mult`.

This holds for `j` as well. The value of `j` defined at line 7 is recorded as reaching to line 9. Moreover, it shows that the value of `j` that reaches to the use at line 9 is defined either at line 4 or at line 5. Even though we can figure out that `j` is defined (except line 7) only at line 5 inside procedure `multi`, the analysis include one more definition point: line 4 inside `sum`. This is because the value of the second parameter `y` (Id 20) (See Figure 5.41), which is bound to the location of `j` at its initial call and is later bound to the location of `i` at its subsequent recursive calls, has an abstract location value that represent these two locations. Thus, when `op2` is called (at line 1 or 3) with the location bound to the second parameter of `f`, the definition expression at line 4 or line 5 records that `i` or `j` are defined at that expression. Note that the procedure pointers bound to the parameters `x` (Id 19) and `y` (Id 20) are either for `sum` (index 3) or `mult` (index 4) (See Figure 5.42).

5.6 Memory Object Lifetime Analysis

In [YH92] we presented an analysis that estimates the memory object lifetime and the temporal locality of memory accesses. Using Z1 we will duplicate the analysis, which was hand-crafted at that time. Again, we present an abstract interpreter which is based upon **Eval1** rather than on **Eval3**, because it will simplify our presentation. An **Eval3**-based specification, which would be a true duplication of [YH93], should not cause any problem.

We estimate, for each object, the intervals between its birth point and its access points. A memory object is a memory location that is allocated for program variables and for "malloc" expressions. The store entry for each location will have a pair of a value `V` and a timer `F`.

```
(domain S (-> Id A))
(domain A (* V F))
```

The location timer is a pair of two "clocks" `P` and `Q`.

```
(domain F (* P Q))
```

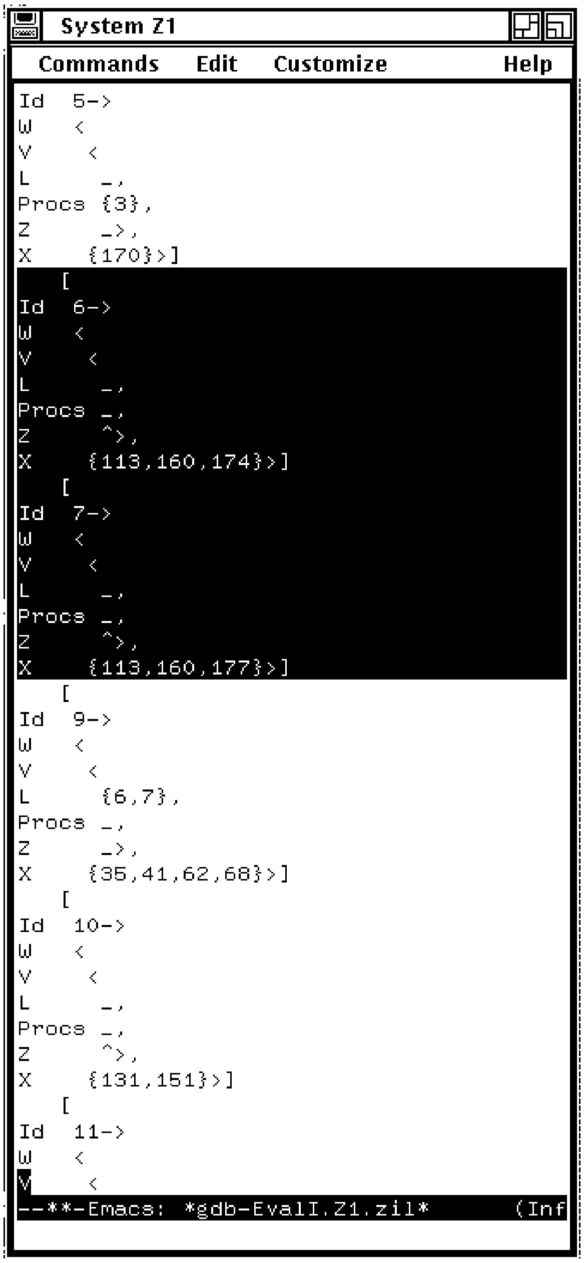


Figure 5.40: Store Image For `i` (Id 6) and `j` (Id 7) At Use Points Of Line 9 In `fptr.c`

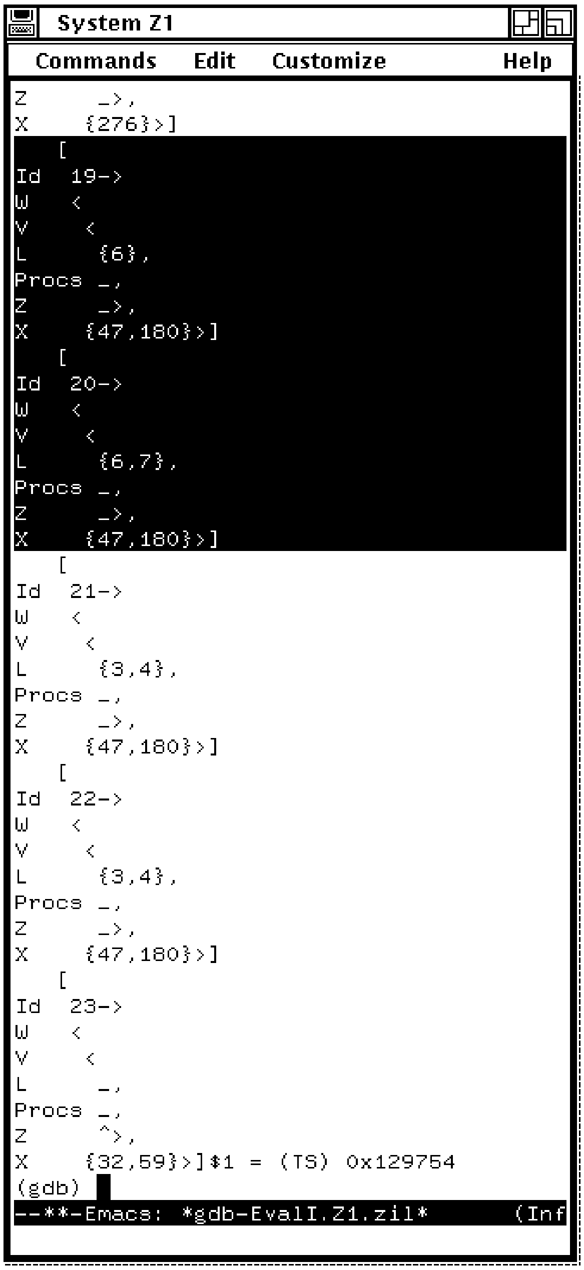


Figure 5.41: Store Image For `x` (Id 19) and `y` (Id 20) In `fptr.c`

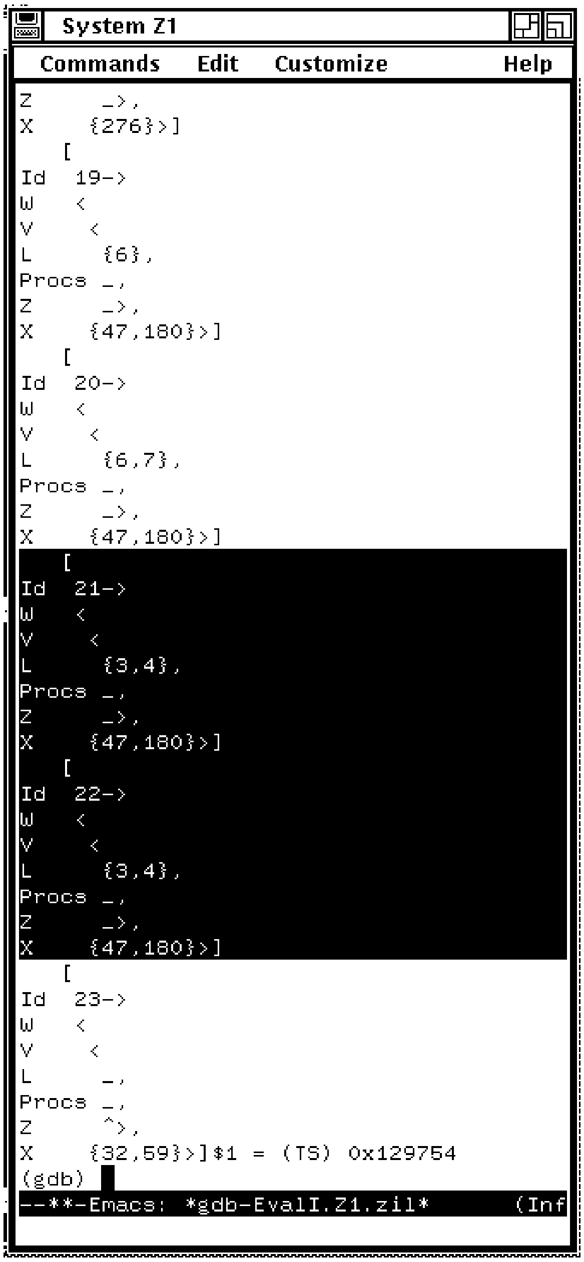


Figure 5.42: Store Image For `op1` (Id 21) and `op2` (Id 22) In `fptr.c`

The first clock `P` starts ticking at the birth of the location. This records the elapsed time since its birth. When this location is read or written, its first clock `P`, which has recorded the elapsed time since its birth, is copied into the second clock. Thus the second clock records the interval that the location is accessed after its birth. What is the time unit? Because our target language MIL does not have iterative constructs other than recursive calls, we choose the procedural movement (procedure call or return) as the time unit. Thus, the clock is a table that maps each procedure to a number of calls and returns (we call these "procedural movements").

`(domain P (-> Proc Gs))`	first clock
`(domain Q (synonym P))`	second clock
`(domain Gs (2^ setG))`	procedural movements

A procedural movement is a tuple of three component `Phi`, `Delf`, and `Delr`.

```
(set setG (* Phi Delf Delr))
```

The first component `Phi` shows whether the procedure is currently active or not (`EPS`), and whether it is a non-recursive call (`D`) or a recursive call (`DD`).

```
(set Phi (elements EPS D DD))
```

The second component `Delf` indicates how many non-recursive calls (flat calls) are invoked and returned. It only has three elements: `f0` means none, `f1` means once, and `ff` means "many times."

```
(set Delf (elements f0 f1 ff))
```

Similarly, the third component `Delr` indicates how may recursive calls are invoked and returned. `r0` means none, `r1` means once, and `rr` means "many times."

```
(set Delr (elements r0 r1 rr))
```

For each program point we examine the store component of the pre-state that is associated with that program point. Each location's first clock shows its age when the control reaches the program point. The intervals between an object's birth time and its access times are also easy to find. Locations are accessed in `read` and `write` expressions in MIL. The store component of the post-state at those expressions have those intervals: they are recorded in the accessed location's second clock.

Several auxiliary functions are defined for manipulating the two clocks of a location.

- `(zeroP (from) (to P))` returns a constant clock `P`, in which each entry for a procedure has `zeroGs`, which represents that the clock is on, but no procedural movements have yet occurred.

- `(zeroF (from) (to F))` returns a constant location timer (a pair of clocks), whose first clock is `(zeroP)`, and whose second clock is bottom. This is the initial clock value for newly allocated memory objects.
- `(stampF (from F) (to F))` copies the first clock of `F` into the second clock. This is to record the object age when it is accessed.
- `(incrDelf (from Delf) (to Delf))` increments the non-recursive call counter. When it is `ff` representing many non-recursive calls, the increment does not change this value. Similarly for `(incrDelr (from Delr) (to Delr))`.
- `(MoveS (from S setProc movement) (to S))`

  When a procedure is called or returned, we need to update the clocks associated with each memory location of the store in order to increment its age. `MoveS` does the job.

Manipulation of the clocks occurs when a location is accessed (read or written) and when a procedural movement occurs (call or return). Evaluation rules for `read` and `write` expressions involve copying the accessed location's first clock into the second clock in order to record the age at the point when it is accessed. The evaluation rule for a `call` expression increments the first clock of each memory location in order to update its age.

- For `read` and `write` expressions, we copy the first clock of the accessed memory location into its second clock. We use the auxiliary function `stampF`.

```
(isRead x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 S))
 (let v1 (! post1 V))
 (let loc (! v1 L))       ; the location to read
 (let s2 (mapjoin (lambda (from setId) (to S) (i)
                    ([/] s1
                         (by (Id i)
                             (A (! (@ s1 (Id  i)) V)
                                ;
                                ; stamp its first clock into the second
                                ;
                                (stampF (! (@ s1 (Id i)) F)))
                             )))
                  loc))
 (let v2 (mapjoin (lambda (from setId) (to V) (i)
                    (! (@ s2 (Id i)) V))
                  loc))
 (POST s2 v2)
 )

(isWrite x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let v1 (! post1 V))
 (let post2 (Eval (NthSubexpr x 2) (! post1 S)))
```

```
(let s2 (! post2 S))
(let v2 (! post2 V))
(let loc (! v1 L))      ; the location to write
(let s3 (mapjoin
         (lambda (from setId) (to S) (id)
           ([/] s2
                (by (Id id)
                    (join
                     (@ s2 (Id id))
                     (A v2
                        ;
                        ; stamp its first clock into the second
                        ;
                        (stampF (! (@ s2 (Id id)) F))
                        )))))
         loc))
(POST s3 v2)
)
```

- For a `call` expression, we increment the first clock of each memory location twice: before we evaluate the procedure body and after we returned from the procedure. This is in order to record the procedure call and its return, respectively. We increment the first clock of each memory location before evaluating the procedure `alpha`'s body by `(MoveS s0 alpha DOWN)`. After returning from the procedure, we increment again the first clock of each memory location by `(MoveS (! post2 S) alpha UP)`. Note also that the parameters and local variables are created by a procedure call, thus their object timers are initialized by `zeroF`.

```
(or (isCall x) (isApply x))
(begin
 ;
 ; evaluate the subexpressions (closure and arguments)
 ...

 ; apply the closure to the evaluated arguments
 ;
 (mapjoin
  (lambda (from setProc) (to POST) (alpha)
    (begin
     (let mom (Body alpha))
     ;
     ; ticks each memory object's first clock because a call happens
     ;
     (let s1 (MoveS s0 alpha DOWN))
     ...
        ;
        ; param binding with new clock (birth) for them
        (let (_s2 0) s1
             (_s2 i) (for i 1 num_param
                        (begin
                         (let param :setId (nth params i))
                         ([/] (_s2 (- i 1))
                              (by (Id param)
                                  (join
                                   (@ (_s2 (- i 1)) (Id param))
                                   (A (_ v (+ i 1))
```

```
#define UNKNOWN (-0x80000000)

int *alloc(int val) {
  int *cell = (int*)malloc(sizeof(int));          (1)
  *cell = val;                                    (2)
  return cell;                                    (3)
}

void h(int *x) {
  *x = *x + 1;
  if (UNKNOWN) h(x);
  else *x = *x - 1;
}

void k(int x) {
  int i, prod;
  prod = 1;
  for (i=0; i < x; i++) prod = prod * i;
}

void g(int *x) {
  int i, sum;
  sum = 0;
  for (i=0; i < *x; i++) sum = sum + i;           (4)
  h(x);                                           (5)
  for (i=0; i < sum; i++) k(i);                   (6)
}

void main() {
 int *obj;

 obj = alloc(10);                    (7)
 g(obj);                             (8)
}
```

Figure 5.43: C program `mem.c`

```
                             ;
                             ; allocated param has new clocks
                             ;
                             (zeroF)))))
                  )))
      ; similarly, initialize locals with new clocks (birth for them)
      ...

      ; evaluate the procedure body
      ;
      (let post2 (Eval mom (_s3 num_local)))
      ;
      ; after the return, ticks each memory object first clock
      ; because a return happens
      ;
      (POST (MoveS (! post2 S) alpha UP) (! post2 V))
      ))))
procs))
```

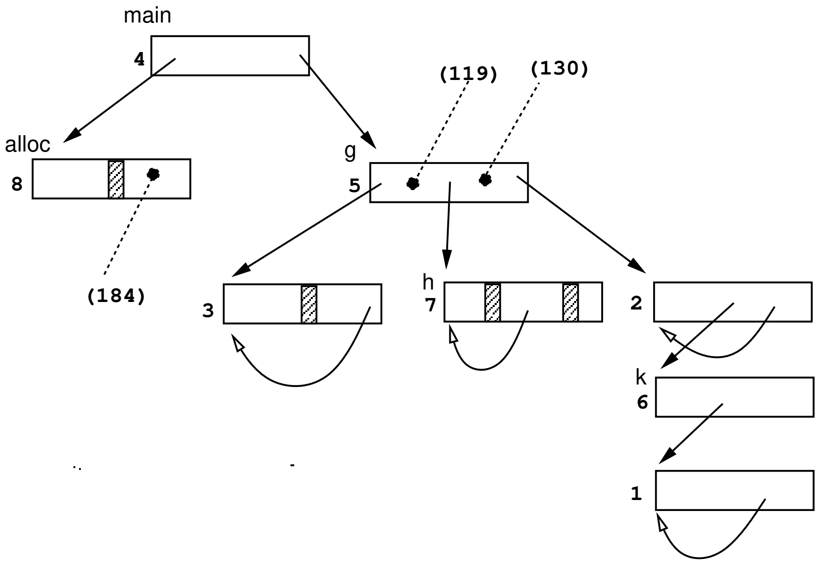


Figure 5.44: The Call Graph of `mem.c`

- **Analysis of `mem.c` (Figure 5.43) With Projection** `(project Z (height> 1))`

Consider the C program in Figure 5.43. At line 7, by procedure call `alloc(10)`, an integer-sized memory block is created and initialized to contain 10, and the new address is assigned to `obj`. After that, the location is passed to another procedure `g`. Inside `g`, the memory object is accessed, at line 4, when we check the iteration upper-bound. After that, its address is passed into another recursive procedure `h` inside which it is written with a new value. After this call to `h` inside `g`, presumably a large computation by procedure `k` is repeatedly invoked, in which no access to the memory object occurs. If we were able to detect that the object's lifetime ended before this computation, then we could explicitly free the machine resource for this object (whether it be a register, a portion of cache, or a portion of the main memory), which may help to improve the subsequent computation's performance.

The call graph of `mem.c` is in Figure 5.44. Rectangle nodes represent the body of each procedure. Execution moves from left to right. The number beside each rectangle is a procedure index. The program starts from the root procedure `main`. Function call is represented by an arrow. A white arrow head indicates that the function call occurs conditionally. This is the case when a function call occurs in one branch of an `if` expression. The shaded areas indicate the program points where the memory object, which is allocated in line 1 of `mem.c`, is accessed. The unnamed rectangles (like 1, 2, and 3) are the procedures that are generated internally for the `for` expressions of the C program (the iterative constructs of

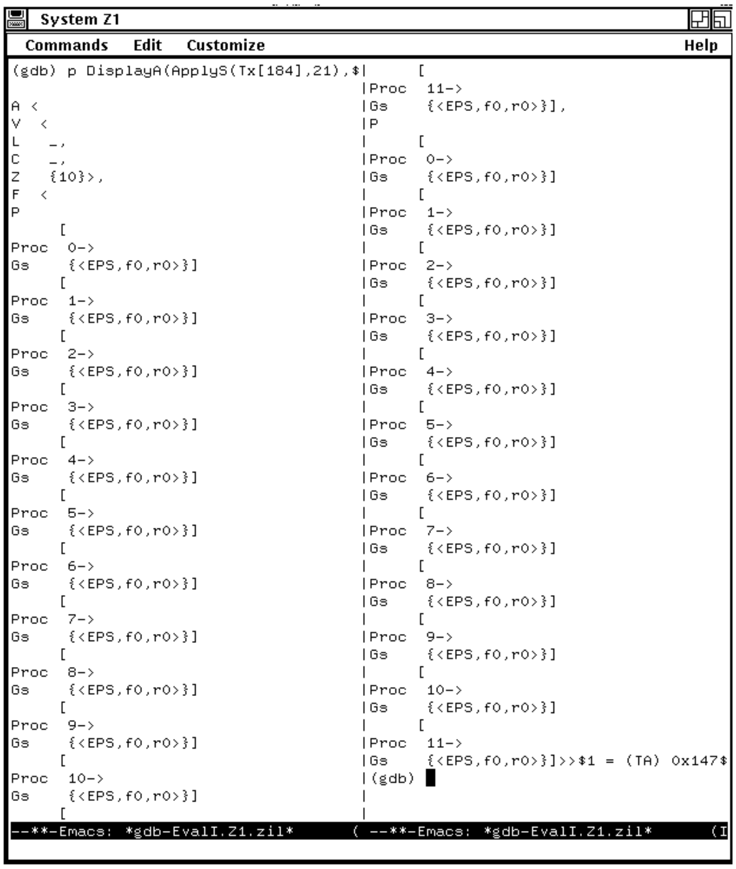


Figure 5.45: Malloc'ed Object Clock Right After Line 2 (Expression 184) of `mem.c`

C are changed into tail-recursive procedure calls in MIL). We will examine the `malloc`'ed object's clocks at three expression points 184, 119, and 130.

The `malloc`'ed object has identifier index 21. When the control reaches expression point 184, object 21 has been born and initialized. But between its birth point and expression 184, no procedural movement happens. Figure 5.45 shows this. The first clock has "zero" time, which indicates no procedure call or return after the creation. The second clock also has "zero" time. This is because when it is accessed inside `alloc` to be initialized to 10, its first clock is copied into its second clock.

Object 21's age at expression 119 is shown in Figure 5.46. We see that procedure 3 (the tail-recursive procedure for the `for` loop in line 4) has experienced one non-recursive call `<EPS, f1, r0>` or one recursive call `<EPS, f0, r1>`. During this lifetime, the last access to object 21 occurs when procedure 3 is called, either initially `<D, f0, r0>` or recursively

`<DD, f0, r0>`. The procedure 3's entry of the second clock of object 21 at expression 119 shows this information. See Figure 5.47.

Object 21's age at expression 130 is shown in Figure 5.48. We see that procedure 3 (tail-recursive procedure for the `for` expression at line 4) has experienced either one non-recursive call `<EPS, f1, r0>` or one recursive call `<EPS, f0, r1>`, procedure 5 (`g`) is currently active `<D, f0, r1>`, procedure 7 (`h`) experienced either one non-recursive call `<EPS, f1, r0>` or one recursive call `<EPS, f0, r1>`.

During this lifespan, the last access to object 21 is recorded in its second clock. See Figure 5.49. It shows the time when procedure 7 (`h`) is active by the initiating call or recursive calls, and that afterwards procedure 3 experienced a flat call or a recursive call.

At the program's last expression point, the object's (Id 21) store entry's second clock contains the time when the last access to the object occurs. This is shown in Figure 5.50. It is the same as its second clock (Figure 5.49) at expression point 130. Since the first clock (Figure 5.48) of the object at that expression point contains an older age than its last access time, we can safely deallocate the object at expression 130.

5.7 Cost Variations by Projections

The tables in Figure 5.51 show the cost variations depending on the projections for **Eval3**. 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 non-linear 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 Z1 (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 Z1 (height> 2))`: `Z`'s elements higher than the two-element sets are projected to top.

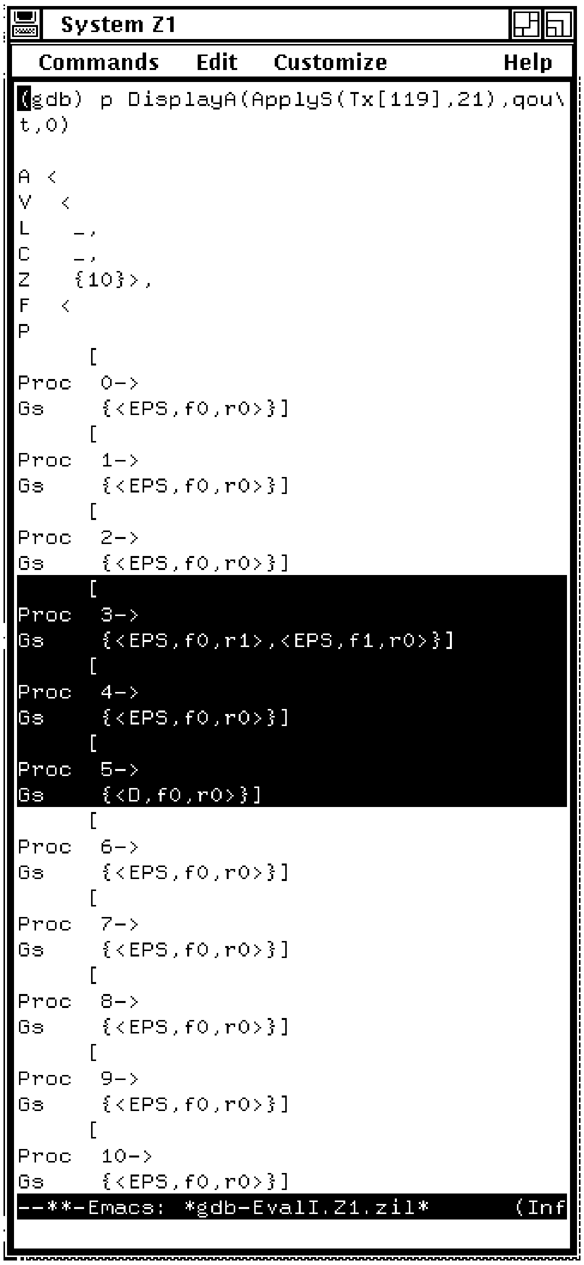


Figure 5.46: Malloc'ed Object's Age Right After Line 4 (Expression 119) of `mem.c`

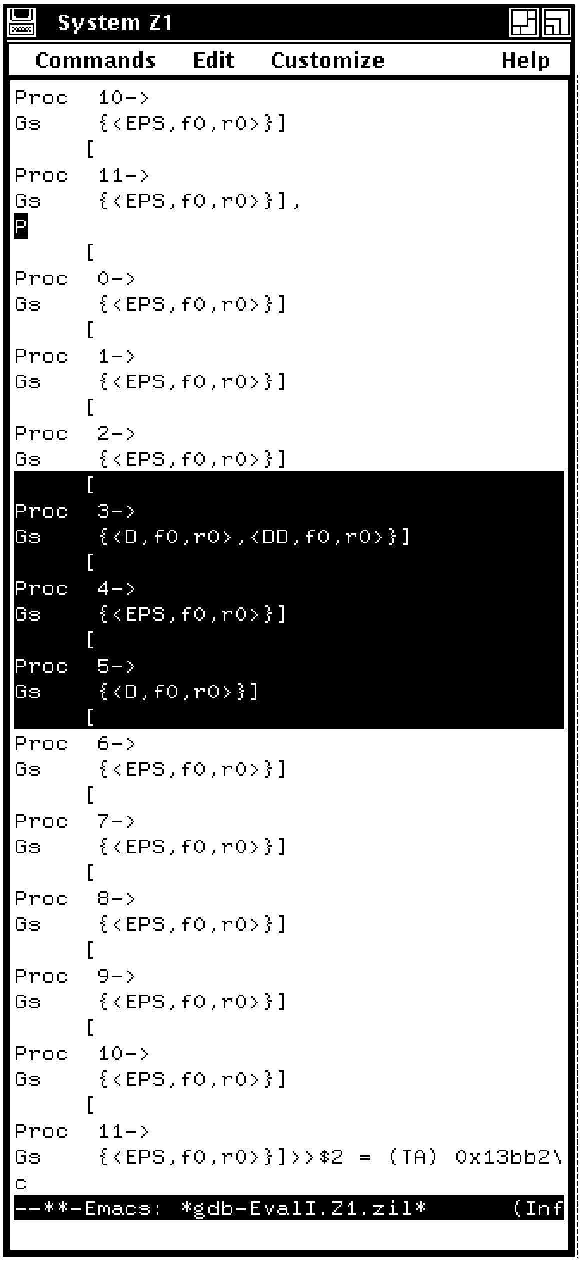


Figure 5.47: Malloc'ed Object's Last Access Time Right After Line 4 (Expression 119) of `mem.c`

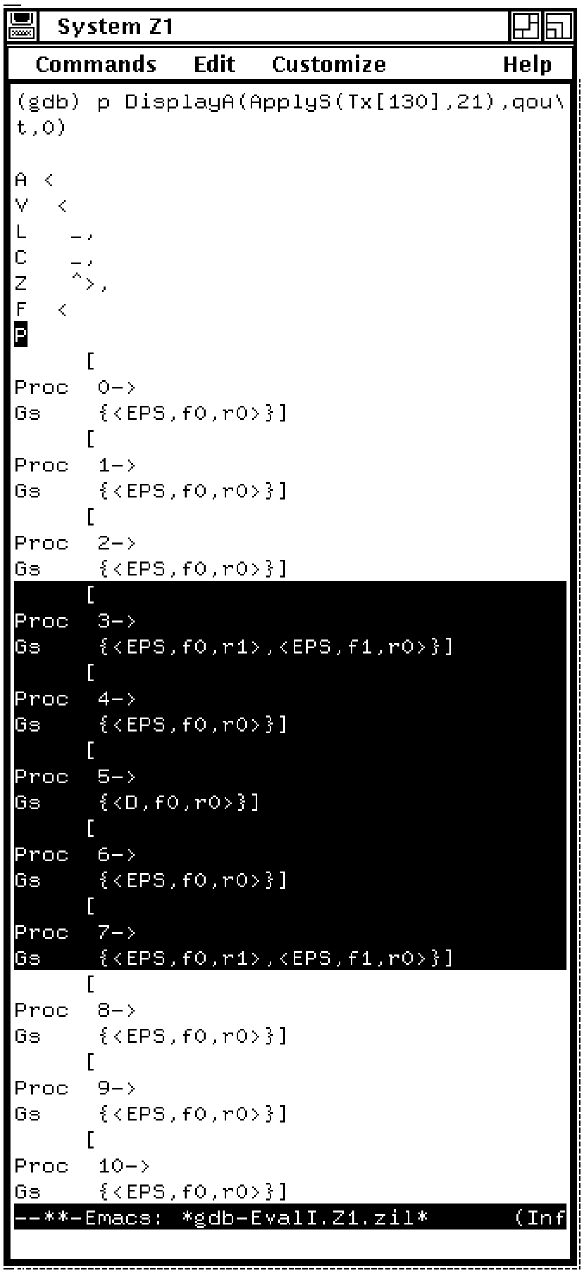


Figure 5.48: Malloc'ed Object's Age Right After Line 5 (Expression 130) of `mem.c`

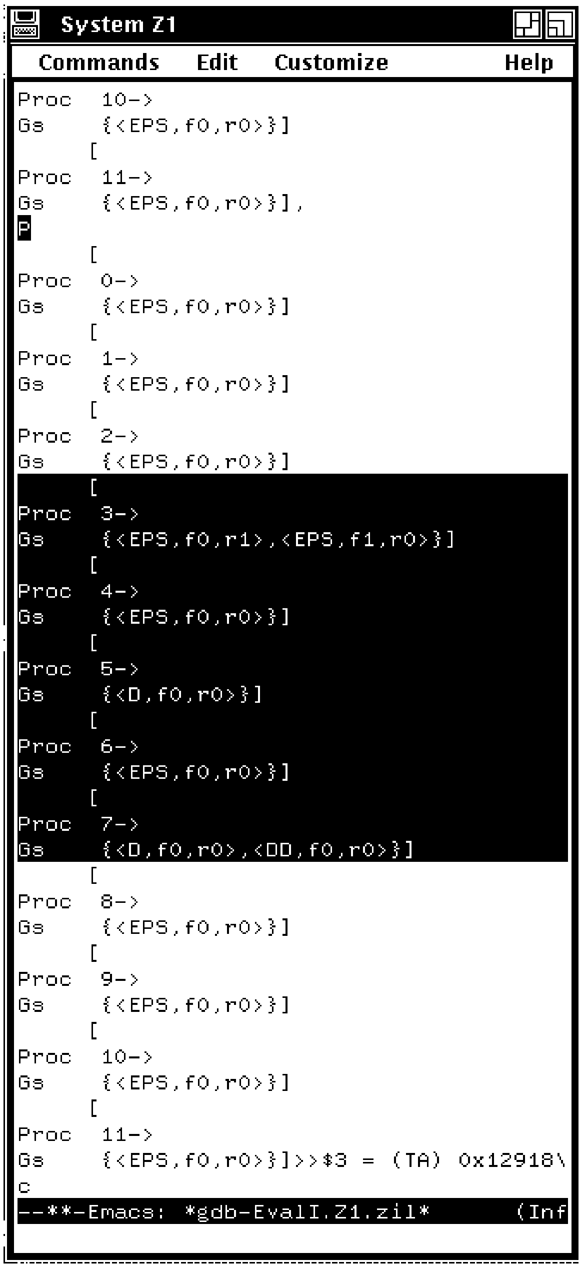


Figure 5.49: Malloc'ed Object's Last Access Time Right After Line 5 (Expression 130) of `mem.c`

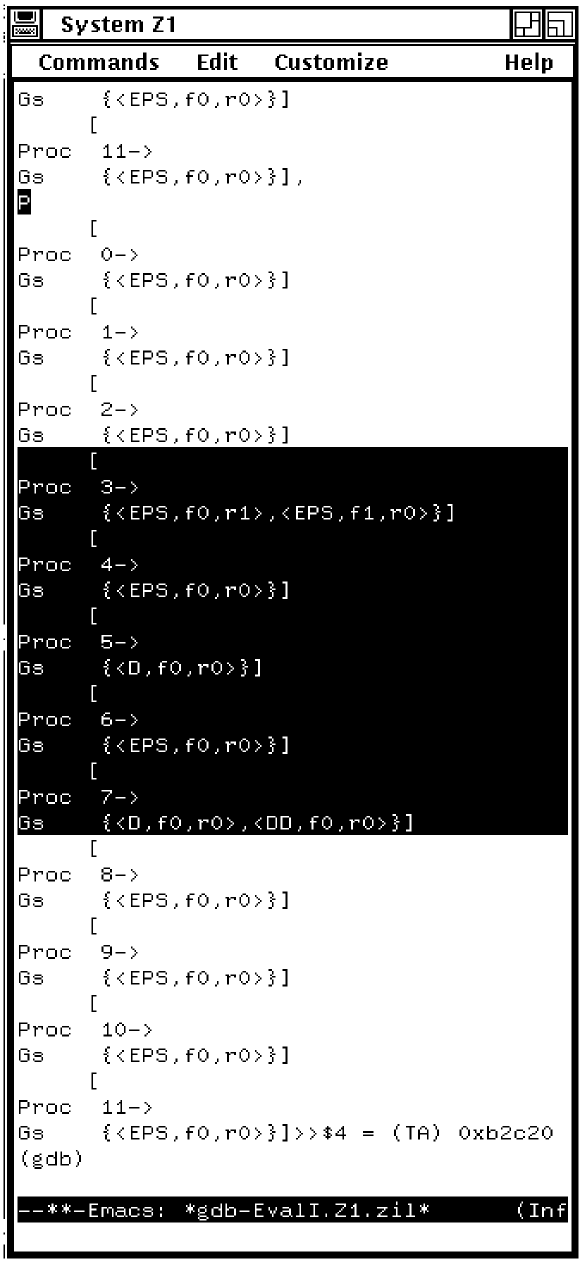


Figure 5.50: Malloc'ed Object's Last Access Time Recorded At The End of The Program `mem.c`

Cost Variations (`wator.c`: 45 procedures, 3467 exprs, 166 vars in MIL)				
Projections	Iterations	CPU sec	MemUse(Garbage)(M)	♯GC
$h(\texttt{Z}) > 1$	340,978	529.21	6.14 (5.27)	94
$h(\texttt{Z}) > 1$ $h(\texttt{Gs}) > 0$	157,678	112.76	7.01 (5.19)	4
$h(\texttt{Z}) > 2$ $h(\texttt{Gs}) > 0$	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
$h(\texttt{Z}) > 1$	376,660	782.29	6.22 (5.29)	186
$h(\texttt{Z}) > 1$ $h(\texttt{Gs}) > 0$	248,721	258.78	6.11 (5.28)	18
$h(\texttt{Z}) > 2$ $h(\texttt{Gs}) > 0$	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
$h(\texttt{Z}) > 1$	784,270	3,647.19	6.72 (5.34)	1,115
$h(\texttt{Z}) > 1$ $h(\texttt{Gs}) > 0$	541,199	519.55	6.42 (5.32)	57
$h(\texttt{Z}) > 2$ $h(\texttt{Gs}) > 0$	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
$h(\texttt{Z}) > 1$	1,084,662	6,134.81	7.01 (5.35)	1,998
$h(\texttt{Z}) > 1$ $h(\texttt{Gs}) > 0$	751,400	969.17	6.42 (5.31)	84
$h(\texttt{Z}) > 2$ $h(\texttt{Gs}) > 0$	779,247	997.14	6.47 (5.32)	93

Figure 5.51: Analysis Cost Variations Depending on Projections

- `(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 average amounts (in megabytes) 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 Z1 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.

5.8 Projection Without Loss of Accuracy

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 will consider such a case in this section.

Let's consider an analysis for finding 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.

We instrument **Eval1** with addition domain `Q`. The interpreter function's pre-state has `Q` component in addition to the store `S`:

```
(domain PRE (* S Q))
```

Post-state also has `Q`:

```
(domain POST (* S V Q))
```

Domain `Q` records, for each procedure, the values bound to its formal parameters:

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

Suppose that whenever one of a procedure's arguments is bound to a non-constant value, 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 $\top_{\mathtt{Z}}$ into $\top_{\mathtt{Arg}}$:

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

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

Consider the program of Figure 5.52. Figure 5.53 shows the analysis result without the projection, and Figure 5.54 shows the case 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. The only difference between these two analyses are their costs. The projected analysis will reach the fixpoint solution faster than the original one.

```
/*
 * 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=l; 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 5.52: Example C Program For Constant Call Analysis

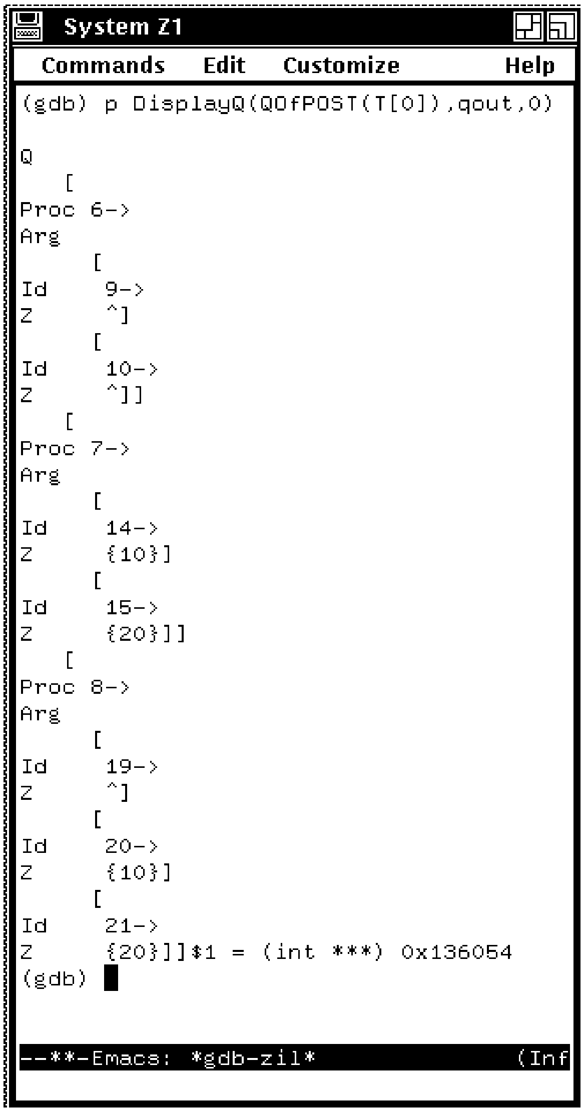


Figure 5.53: Constant Call Analysis Without the Projection

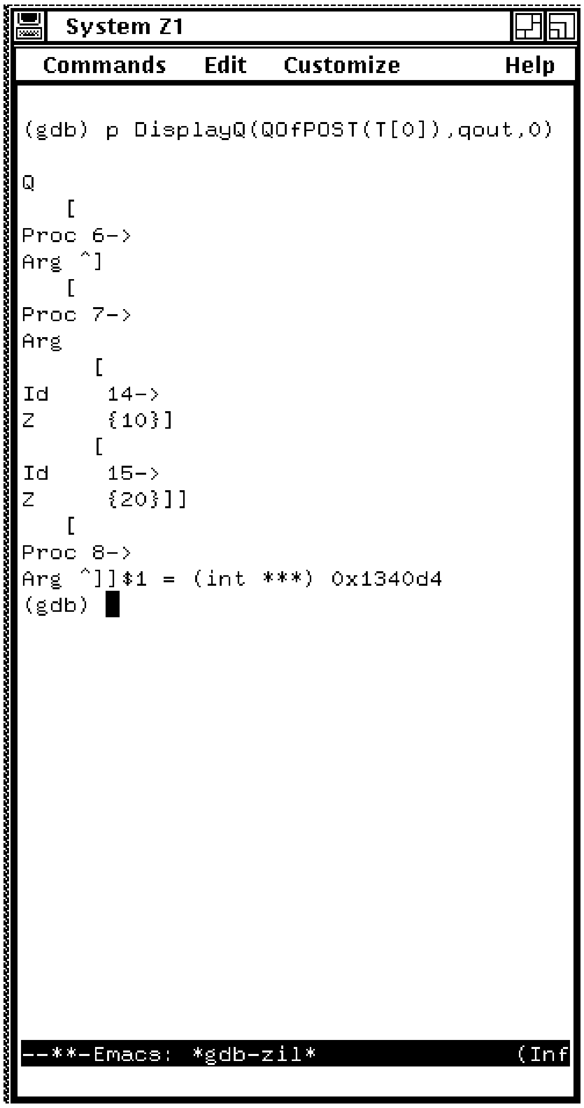


Figure 5.54: Constant Call Analysis With the Projection

Chapter 6

Conclusion

Designing a program analysis is difficult because it involves various parameters. In particular, one must carefully measure the accuracy of the analysis against the memory and cpu time that are required to compute it. To reach an acceptable cost-accuracy balance has generally meant to repeatedly re-implement the analysis, measure its behavior, alter its design, and so on.

In this thesis we present a software system called system Z1, by which one can quickly specify a program analysis, automatically obtain from this specification an executable version, measure its performance, easily alter the design (adjust the cost-accuracy balance), and repeat the cycle until the desired behavior is observed. The specification language Z1 is designed so that the user can conveniently express various program analyses for a wide range of target languages. The space of domains (lattices) that can be defined in Z1 is closed in the sense that Z1's domain constructors accepts any domain in the space and builds new one which is still contained in the space. Even though it remains to be seen whether the specification language is general enough to specify a large class of program analyses for a wide range of target languages, and is abstract enough to specify an analysis succinctly, our experience with it suggests that is it. As we try more analyses, we plan to extend the language if necessary.

What makes system Z1 especially useful is called a *projection expression*. The user writes down, at the outset, a straightforward specification of an analysis, essentially ignoring the cost (memory and cpu time) of computing it. Afterwards, projection expressions are used to control its cost-accuracy tradeoff. A projection is a map that simplifies a domain over which an abstract interpreter is defined. The user defines projections for selected domains. System Z1 then automatically generates a new analyzer which will conduct the analysis over the simplified domain structure. Projection expressions permit accuracy to be traded against cost, without requiring that the specification of the analysis be altered.

We demonstrated Z1's capabilities by experiments with a series of generated analyzers.

We showed how we can use projection expressions to change the accuracy-cost balance of the analyses. Although it is possible to use the system to analyze any programming language, we use it to analyze MIL (MIPRAC [Har92, HA92] Intermediate Language) programs, because we have C, FORTRAN, and SCHEME translators to MIL already built.

Z1 produces a C program from the specification; the program becomes an executable after it is compiled and linked with a fixed point algorithm, garbage collector, various debugging aids, and lattice element pretty printers. These modules are produced by Z1 system.

We can summarize the highlights of Z1 as follows.

- By System Z1, one can quickly implement an analysis, measure its performance, and manage its cost-accuracy tradeoff. It generates an analysis (a C code) from a high-level abstract interpreter definition.
- System Z1 has a simple mechanism to tune the cost-accuracy tradeoff. These are called *projection expressions*. They are defined over the types of an abstract interpreter, and simplify its structure.
- System Z1 can be used as a fast prototyping tool for complex program analyses for C, FORTRAN, and SCHEME programs. We can design a variety of program analyses for these languages using Z1.
- System Z1 facilitates the use of abstract interpretation for imperative as well as functional programming languages.

6.1 Related Works

Almost every compiler-component has a corresponding automatic tool: LEX [ES86] for the lexical analyzer, YACC [Joh86] and PGS [GK88] for the parser, TWIG [AGT89] and [NN, App85, Ras82] for the code generator, LIGA [Kas90] and [Far86] for the attribute grammar evaluator, and Sharlit [TH92] and GOSpeL/GENesis[WS91] for the code optimizer. Collections of compiler-component generators have been integrated into single compiler writing systems: ELI [GHLaWMW92], MUG2 [Wil81, GG82], HLP84 [KNPS88], and Rie/Jun [Sas90]. Moreover, based on the observation that compiling a program amounts to emitting a representation of its meaning as object code, a rich set of semantics-directed compiler generation tools are reported: Mess [LP87, Lee89], Ceres [Tof90], PERLUETTE [Gau80]), and [Mos76, Mos80, Pau82, NN86, CJ83, JS80, HM90]

We have added system Z1 (a program analyzer generator) to this collection. Z1 is distinguished from other similar tools [TH92, Sas90, KNPS88, GG82, Wil81] in three ways. First, system Z1 is based on the abstract interpretation framework. Because of this, (as we discussed in Section 2.1.1), the user can define comfortably a correct, detailed program analyses (whether it be interprocedural or intra-procedural) for difficult and dynamic programming languages as are found in C and SCHEME. The input is an abstract semantics of the target language which is expressed as a definitional interpreter [Rey72]. Second, it has a convenient mechanism (projection expression) by which the user can control the accuracy and cost tradeoff of the analysis. Projections over the types of an abstract interpreter permits high-level management of the accuracy-cost tradeoff of the analysis. Third, it has a high-level specification language in which the user can conveniently express an abstract interpreter succinctly and conveniently.

Other analysis generation tools [TH92, Sas90, KNPS88, GG82, Wil81] have difficulty in handling interprocedural analysis specifications, are limited to specific target languages, are limited to a class of simple analyses (e.g. bit-vector representation of the data flow values), require substantial amount of code (in the implementation language) 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.

Tjiang and Hennessy [TH92] reported a tool called Sharlit which generates program analyzers and program optimizer. Its specification language is C++. An interesting feature of this tool is that its fixpoint computation procedure can be simplified by means of flow graph reduction rules. This assumes that the flow graph of the program being analyzed are available prior to the analysis. It is unclear, however, how this feature can be used for the case where the flow graph of a program must be derived during analysis. In addition to analysis specification, Sharlit allows optimization specification, that system Z1 lacks. Young [You89] reported a library that he used to implement several semantic analyses. Steffen [Ste91] reported a specification framework that uses modal logic formulae to specify data flow analysis algorithms. Venkatesh [Ven89] defines a specification language that 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. Whitfield and Soffa [WS91] reported an optimizer generator called GOSpeL/GENesis. The user specifies an optimization preconditions and actions; analysis result of the input program is assumed. Combined with Z1 with a proper interface, it would provide a full automation of the optimization phase of a compiler system.

6.2 Limitations and Discussion

System Z1 is not a perfect tool. It has several limitations.

- **Correctness.** System Z1 does not provide any assistance for designing a correct abstract interpreter; it only checks its type correctness. For example, it does not check the monotonicity of the functions in the abstract interpreter. Its input is assumed to be a correct abstract interpreter. The user should refer to one of the frameworks (as summarized in [CC92b, CC92a]) for defining correct abstract interpreters, which are based either on denotational semantics [Bur87, Bur91, MJ85, Nie85] or on operational semantics [CC77, CC79, CC92b]. In Section 2.1.2 we discussed one approach based on denotational semantics.

- **Completeness of Projections.** Projection is not a complete notion for controlling the accuracy-cost tradeoff of an abstract interpreter. There can exist two related abstract interpreters (in terms of accuracy and cost), such that one is not a projected image of the other. This means that there may be a case in which the user has to give up applying projections to an existing abstract interpreter because, by doing so, he probably cannot acquire a satisfying one that has an acceptable accuracy-cost tradeoff.

  Another interesting question is this: how easy would it be to find, given an elaborate abstract interpreter, projections that minimize the accuracy loss while maximizing the cost reduction?

- **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.

- **Separate and Incremental Analysis.** The generated analyzer requires the whole program to reside in a single file, and it analyzes from scratch. When the source program is distributed across several files, separate analysis of each file and later merging their results would be a more practical approach for real programs. Also, when the source program is changed slightly, updating only the necessary parts of the previously computed analysis result, instead of starting anew, would reduce substantially the analysis cost for the modified program. It would be nice to make system Z1 generate such intelligent program analyzer.

- **Theorem Generator.** 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 could provide a specification language in which the user can write rules for data analysis.

- **User Specified Fixpoint Computation Optimization.** We have used a general fixpoint computation method. Depending on the target language, 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 [TH92], system

Z1 can provide a facility by which the user can specify such optimizations when possible.

- **Optimization of the Generated Code.** There are ways in which we can further improve the performance of the generated C code. For example, domain elements are currently 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.

Bibliography

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[Amm92] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. on Software Engineering*, 18(3):237–251, March 1992.

[App85] Andrew W. Appel. Semantics-directed code generation. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 315–324, 1985.

[Bar77] Jeffrey M. Barth. An interprocedural data flow analysis algorithm. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 119–131, January 1977.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction (SIGPLAN Notice Vol.21, No.7)*, pages 162–175, July 1986.

[BH91] B.A.Davey and H.A.Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Texbooks. Cambridge University Press, 1991.

[Bur87] Geoffrey L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Department of Computing, Imperial College, University of London, March 1987.

[Bur91] Geoffrey L. Burn. The abstract interpretation of higher-order functional languages: From properties to abstract domains (technical summary).

Technical report, Imperial Collge of Science, Technology and Medicine, 1991.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–56, 1988.

[CC77] 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.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. Technical Report LIX/RR/92/10, Ecole Polytechnique, 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, 1992.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, 1991.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *SIGPLAN Notices*, 21(7):152–161, July 1986.

[CH92] Liling Chen and Williams Ludwell Harrison III. An efficient algorithm for complex program analysis. Technical report, Center for Supercomputing R & D, Univ. of Illinois at Urbana-Champaign, 1992.

[CJ83] Henning Christiansen and Neil Jones. Control flow treatment in a simple semantics-directed compiler generator. In D. Bjorner, editor, *Formal Description of Programming Concepts II*, pages 73–97. North-Holland Publishing Company, 1983.

[CR88] Martin D. Carroll and Barbara G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 274–284, 1988.

[Eor91] Istvan Eorgacs. Occurrence-sensitive interprocedural data flow analysis. In *Proceedings of the Second Symposium on Programming Languages and Software Tools*, pages 55–69, August 1991.

[ES86] M. E.Lesk and E. Schmidt. Lex: A lexical analyzer generator. In *Unix Programmer's Manual Supplementary Documents 1*. 1986.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of SIGPLAN '86 Symposium on Compiler Construction*, SIGPLAN Notices, pages 85–98, 1986.

[Gau80] M. C. Gaudel. Specification of compliers as abstract data type representation. In *Lecture Notes in Computer Science*, volume 94, pages 140–164, 1980.

[GG82] 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.

[GHLaWMW92] R. W. Gray, V. P. Heuring, S. P. Levi, and A. M. Sloane ans W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2), 1992.

[GK88] J. Grosch and E. Klein. User manual for the pgs system. Technical report, GMD, Research Institute at the University of Karlsruhe, 1988.

[GKK+80] G.Giertz, K.H.Hofmann, K.Keimel, J.D.Lawson, M.Mislove, and D.S.Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, 1980.

[GW76] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, 1976.

[HA92] 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.

[Har89] Williams L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3 and 4):179–396, 1989.

[Har92] Williams Ludwell Harrison III. *Semantic Analysis of Symbolic Programs for Automatic Parallelization*. Book in preparation, 1992.

[Hec77] Matthew S. Hecht. *Flow Analysis of Comptuer Programs.* Elsevier North-Holland, Inc., 1977.

[HK92] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. Technical Report COMP TR92-187, Rice University, July 1992.

[HM90] John Hannan and Dale Miller. From oeprational semantics to abstract machines: Preliminary results. In *Proceedings of the SIGPLAN Conference on Lisp and Functional Programming*, pages 323–332, 1990.

[HY91] Paul Hudak and Jonathan Young. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13(2):269–290, April 1991.

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Joh86] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual Supplementary Documents 1.* 1986.

[JS80] Neil D. Jones and David A. Schmidt. Compiler generation from denotational semantics. In *Lecture Notes in Computer Science*, volume 94, pages 70–93, 1980.

[Kas90] U. Kastens. Liga: A language independent generator for attribute evaluator. Technical report, University of Paderborn, 1990.

[Ken76] Ken Kennedy. A comparison of two algorithms for global data flow analysis. *SIAM Journal on Computing*, 5(1):158–180, March 1976.

[KNPS88] 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.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative alorithm. *Journal of the ACM*, 23(1):158–171, 1976.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 77.

[Lee89] Peter Lee. *Realistic Compiler Generation.* Foundations of Computing Series. The MIT Press, 1989.

[LP87] Peter Lee and Uwe Pleban. A realistic compiler generator based on high-level semantics. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 284–295, 1987.

[MJ85] Alan Mycroft and Neil D. Jones. A relational framework for abstract interpretation. In *Lecture Notes in Computer Science*, volume 217. Springer-Verlag, 1985.

[Mos76] P. D. Mosses. Compiler generation using denotational semantics. In *Lecture Notes in Computer Science*, volume 76, pages 436–441. Springer-Verlag, 1976.

[Mos80] Peter Mosses. A constructive approach to compiler correctness. In *Lecture Notes in Computer Science*, volume 94, pages 189–210. 1980.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, 1990.

[Mye81] Eugene W. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[Nie85] Flemming Nielson. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, July 1985.

[NN] Flemming Nielson and Hanne R. Nielson. Code generation from two-level denotational meta-languages. In *Lecture Notes in Computer Science*, volume 217, pages 192–205. Springer-Verlag.

[NN86] Hanne R. Nielson and Flemming Nielson. Semantics directed compiling for functional languages. In *Proceedings of the SIGPLAN Conference on Lisp and Functional Programming*, pages 249–257, 1986.

[Pau82] Lawrence Paulson. A semantics-directed compiler generator. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 224–233, 1982.

[Ras82] Martin R. Raskovsky. Denotational semantics as a specification of code generators. In *Proceedings of SIGPLAN '82 Symposium on Compiler Construction*, volume 17 of *SIGPLAN Notices*, pages 230–244, 1982.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, August 1972.

[Ros77] Barry K. Rosen. Applications of high level control flow. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 38–47. CACM, January 1977.

[Ros80] Barry K. Rosen. Monoids for rapid data flow analysis. *SIAM Journal on Computing*, 9(1), February 1980.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[Rut65] D.E. Rutherford. *Introduction to Lattice Theory.* Hafner Publishing Company, New York, 1965.

[Sas90] Masataka Sassa. Rie and jun: Towards the generation of all compiler phases. In *Lecture Notes in Computer Science*, volume 477, pages 56–70. 1990.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. In *Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems.* IEEE, 1970.

[Sco72] Dana S. Scott. Mathematical concepts in programming language semantics. In *AFIPS Conference Proceedings*, volume 40, pages 225–234, 1972.

[Shi88] Olin Shivers. Control flow analysis in scheme. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, May 1991.

[Ste91] Bernahard Steffen. Data flow analysis as model checking. In *Lecture Notes in Computer Science*, volume 526, pages 346–364. 1991.

[Sto77] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Tar76] Robert Endre Tarjan. Iterative algorithms for global flow analysis. Technical Report Stan-cs-76-547, Stanford University, March 76.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit - a tool for building optimizers. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 82–93, June 1992.

[Tof90] Mads Tofte. *Compiler Generators: What They Can do, What They Might Do, and What They Will Probably Never Do.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1990.

[Ven89] G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1989.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, 1980.

[Wil81] 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.

[WS91] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 120–129, June 1991.

[YH92] Kwangkeun Yi and Williams Ludwell Harrison III. Interprocedural data flow analysis for compile-time memory management. Technical Report 1244, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, August 1992.

[YH93] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 246–259 (also as CSRD Report No. 1260), January 1993.

[You89] Jonathan H. Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages*. PhD thesis, Yale University, May 1989.

Appendix A

Availability

System Z1 is available for `ftp`[1] Connect to `sp2.csrd.uiuc.edu`

```
%ftp sp2.csrd.uiuc.edu
```

Login as "anonymous" with your login name as password, and, on the ftp prompt, change directory into `pub/kwang`

```
ftp> cd pub/kwang
```

All sources and utility files are available in a compressed tar file

```
Z1.tar.Z
```

All the test programs and analysis specifications that we used in Chapter 5 are also provided in the tar file.

[1] File Transfer Program.

Appendix B

Specification for Eval1

```
; the most primitive abstract interpreter Eval1.
; instance-insensitive and array and structure variable has sinlge location
; in the store

(types
 (domain PRE (synonym S))
 (domain POST (* S V))
 (domain S (-> Id V))
 (domain V (* L C Z))
 (domain L (2^ setId))      ;(project L (height> 1))
 (domain C (synonym Procs))
 (domain Ids (2^ setId))
 (domain Id (flat setId))
 (domain Procs (2^ setProc))
 (domain Z (2^ setZ))       (project Z (height> 2))
 (domain T (flat setB))
 (domain Proc (flat setProc))
 (set setId (index NumOfPlaces))
 (set setProc (index NumOfProcs))
 (set setB (elements True False))
 (set setZ (range -999999 999999))        ;-infinity to +infinity
)

; parser interface declarations
;(ExprSize (from syntree) (to setY))
;(PlaceSize (from syntree) (to setY))
(NumOfExprs NumOfProcs NumOfPlaces (from) (to number))
(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))
;(NthParamSize (from setProc number) (to setY))
;(NthLocalSize (from setProc number) (to setY))
;(LSize CSize ZSize (from) (to setY))
(Owner (from setId) (to setProc))
(DynamicId (from syntree) (to setId))
(ProcId (from syntree) (to setProc))
(Xid (from syntree) (to setId))
(Xnum (from syntree) (to number))
```

```
(Body (from setProc) (to syntree))
(isBegin isIf isRead isWrite isCreate isCall isApply isProc isClosure
 isId isNumber isPlus isMinus isMulti isMod isDiv
 isEq isLe isLt isGe isGt isNot isAnd isOr isEtcBiOp isEtcUnOp
 (from syntree) (to number))

;
; auxiliary semantic functions
;
(numToZ (from number) (to Z))
(function numToZ (n)
          (if (or (< n -99999) (> n 99999)) (Z top)
            (Z (setZ n))))

(plusZ (from Z Z) (to Z))
(function plusZ (z1 z2)
          (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
            (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
              (mapjoin (lambda (from setZ) (to Z) (setz1)
                         (mapjoin (lambda (from setZ) (to Z) (setz2)
                                    (numToZ (+ (number setz1)
                                                (number setz2))))
                                  z2))
                       z1))))

(minusZ (from Z Z) (to Z))
(function minusZ (z1 z2)
         (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
           (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
             (mapjoin (lambda (from setZ) (to Z) (setz1)
                        (mapjoin (lambda (from setZ) (to Z) (setz2)
                                   (numToZ (- (number setz1)
                                               (number setz2))))
                                 z2))
                      z1))))

(multiZ (from Z Z) (to Z))
(function multiZ (z1 z2)
          (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
            (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
              (mapjoin (lambda (from setZ) (to Z) (setz1)
                         (mapjoin (lambda (from setZ) (to Z) (setz2)
                                    (numToZ (* (number setz1)
                                                (number setz2))))
                                  z2))
                       z1))))

(numTosetB (from number) (to setB))
(function  numTosetB (n)  (if (= n 0) False True))

(ZtoT (from Z) (to T))
(function ZtoT (z)
          (if (= z (Z bottom)) (T bottom)
            (if (= z (Z top)) (T top)
              (if (<= z (Z (setZ 0)))        ; is z the singleton set {0}
                  (T False)
                ; z is a set of setZ elements
                (if (in (setZ 0) z) (T top) (T True))))))

(TtoZ (from T) (to Z))
```

```
(function TtoZ (t)
          (if (= t (T bottom)) (Z bottom)
            (if (= t (T top)) (Z top)
              (if (= t (T False)) (Z (setZ 0))
                (Z (setZ 1))))))

; abstract integer equal
(equalZ (from Z Z) (to T))
(function equalZ ( z1 z2)
      (if (or (= z1 (Z bottom))
              (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top))
                (= z2 (Z top))) (T top)
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (= (number setz1)
                                         (number setz2)))))
                                z2))
                     z1)))))

; abstract integer less than equal
(leZ (from Z Z) (to T))
(function leZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (<= (number setz1)
                                          (number setz2)))))
                                z2))
                     z1)))))

; abstract integer less than
(ltZ (from Z Z) (to T))
(function ltZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T False)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (< (number setz1)
                                         (number setz2)))))
                                z2))
                     z1)))))

; abstract integer greater than equal
(geZ (from Z Z) (to T))
(function geZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (<= (number setz2)
```

```
                                        (number setz1)))))
                              z2))
                    z1)))))

; abstract integer greater than
(gtZ (from Z Z) (to T))
(function gtZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T False)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                      (mapjoin (lambda (from setZ) (to T) (setz2)
                                (T (numTosetB
                                    (> (number setz1)
                                       (number setz2)))))
                              z2))
                    z1)))))

; abstract bool and
; note that when first is false, return false
(andT (from T T) (to T))
(function andT (t1 t2)
          (if (= t1 (T False)) t1
            (if (= t1 (T True)) t2
              (if (= t1 (T bottom)) t1
                (T top)))))

; abstract bool or
; note that when first is true, return true
(orT (from T T) (to T))
(function orT (t1 t2)
          (if (= t1 (T True)) t1
            (if (= t1 (T False)) t2
              (if (= t1 (T bottom)) t1
                (T top)))))

; abstract bool not
(notT (from T) (to T))
(function notT (t)
          (if (= t (T bottom)) (T bottom)
            (if (= t (T top)) (T top)
              (if (= t (T True)) (T False)
                (T True)))))

(equalL (from L L) (to T))
(function equalL (l1 l2)
          (if (or (= l1 (L bottom)) (= l2 (L bottom))) (T bottom)
            (if (or (= l1 (L top)) (= l2 (L top))) (T top)
              (if (= l1 l2) (T True) (T False)))))
(equalC (from C C) (to T))
(function equalC (c1 c2)
          (if (or (= c1 (C bottom)) (= c2 (C bottom))) (T bottom)
            (if (or (= c1 (C top)) (= c2 (C top))) (T top)
              (if (= c1 c2) (T True) (T False)))))


;
; The semantics function
;

(Eval (from syntree PRE) (to POST))        (start (S bottom))
```

```
(semantics Eval (x s)
  (switch
   (isBegin x)
   (begin
    (let (_post 1) (Eval (NthSubexpr x 1) s)
         (_post i) (for i 2 (NumOfSub x)
                        (Eval (NthSubexpr x i) (! (_post (- i 1)) S)
                             )))
    (_post (NumOfSub x))
    )

   (isIf x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) s))
    (let s1 (! post1 S))
    (let v1 (! post1 V))
    (let cond (equalZ (! v1 Z) (Z (setZ 0))))
    (let ppost2 (if (<= (T False) cond)
                    (Eval (NthSubexpr x 2) s1)
                  (POST bottom)))
    (let ppost3 (if (<= (T True) cond)
                    (Eval (NthSubexpr x 3) s1)
                  (POST bottom)))
    (join ppost2 ppost3)
    )

   (isRead x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) s))
    (let s1 (! post1 S))
    (let loc (! (! post1 V) L))                ;location value
    (let v1 (mapjoin (lambda (from setId) (to V) (i)
                       (@ s1 (Id i)))
                     loc))
    (POST (! post1 S) v1)
    )

   (isWrite x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) s))
    (let loc (! (! post1 V) L))
    (let post2 (Eval (NthSubexpr x 2) (! post1 S)))
    (let s2 (! post2 S))
    (let v2 (! post2 V))
    (let s3 (mapjoin
             (lambda (from setId) (to S) (id)
               ([/] s2
                    (by (Id id)
                        (join (@ s2 (Id id)) v2))))
             loc))
    (POST s3 v2)
    )

   (isCreate x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) s))
    (let s1  (! post1 S))
    (let id  (DynamicId x))
    (let v (V (L id) (C bottom) (Z bottom)))
    (POST s1 v)
```

```
 )

(or (isProc x) (isClosure x))
(POST s (V (L bottom)
           (C (Procs (ProcId x)))
           (Z bottom)))

(isId x)
(begin
 (let id (Xid x))
 (POST s (V (L id) (C bottom) (Z bottom)))
 )

(isNumber x)
(POST s (V (L bottom) (C bottom) (numToZ (Xnum x))))

(or (isCall x) (isApply x))
(begin
 (let n (NumOfSub x))
 (let num_actual (- n 1))
 (let (_post 1) (Eval (NthSubexpr x 1) s)
      (_post i) (for i 2 n
                     (Eval (NthSubexpr x i)
                           (! (_post (- i 1)) S))))
 (let (_v i) (for i 1 n (! (_post i) V))) ;closure and actual params
 (let s1 (! (_post n) S))
 (let procs (! (_v 1) C))
 (mapjoin
  (lambda (from setProc) (to POST) (alpha)
     (begin
     (let mom (Body 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)))))
        (let (_s2 0) s1        ; param binding
             (_s2 i) (for i 1 num_param
                          (begin
                           (let param :setId (nth params i))
                           ([/] (_ s2 (- i 1))
                                (by (Id param)
                                    (join (@ (_ s2 (- i 1)) (Id param))
                                          (_ v (+ i 1)))))
                           )))
        ; initialize local
;        (let num_local (NumOfLocal alpha))
;        (let locals :Ids (join (for i 1 num_local
;                                    (Ids (NthLocal alpha i)))))
;        (let (_s3 0) (_s2 num_param)
;             (_s3 i) (for i 1 num_local
;                          (begin
;                           (let local :setId (nth locals i))
;                           ([/] (_s3 (- i 1))
;                                (by (Id local)
;                                    (join (@ (_ s3 (- i 1)) (Id local))
;                                          (V (L bottom)
;                                             (C bottom)
;                                             (Z (setZ 0))))))
```

```
;                              )))
           (Eval mom (_ s2 num_param))
           ))
       ))
      procs
      ))

   (isPlus x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) s))
    (let s1 (! post1 1))
    (let v1 (! post1 2))
    (let post2 (Eval (NthSubexpr x 2) s1))
    (let s2  (! post2 1))
    (let v2  (! post2 2))
    (let v (V (join (! v1 L) (! v2 L))
              (C bottom)
              (plusZ (! v1 Z) (! v2 Z))))
    (POST s2 v)
    )

   (isMulti x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) s))
    (let s1 (! post1 1))
    (let v1 (! post1 2))
    (let post2 (Eval (NthSubexpr x 2) s1))
    (let s2  (! post2 1))
    (let v2  (! post2 2))
    (let v (V (L bottom)
              (C bottom)
              (multiZ (! v1 Z) (! v2 Z))))
    (POST s2 v)
    )

   (isMinus x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) s))
    (let s1 (! post1 1))
    (let v1 (! post1 2))
    (let post2 (Eval (NthSubexpr x 2) s1))
    (let s2  (! post2 1))
    (let v2  (! post2 2))
    (let v (V (L bottom)
              (C bottom)
              (minusZ (! v1 Z) (! v2 Z))))
    (POST s2 v)
    )

   (isLt x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) s))
    (let s1 (! post1 1))
    (let v1 (! post1 2))
    (let post2 (Eval (NthSubexpr x 2) s1))
    (let s2  (! post2 1))
    (let v2  (! post2 2))
    (let v (V (L bottom)
              (C bottom)
              (TtoZ (ltZ (! v1 Z) (! v2 Z)))))
```

```
 (POST s2 v)
 )

(isLe x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (leZ (! v1 Z) (! v2 Z)))))
 (POST s2 v)
 )

(isGe x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (geZ (! v1 Z) (! v2 Z)))))
 (POST s2 v)
 )

(isGt x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (ltZ (! v2 Z) (! v1 Z)))))
 (POST s2 v)
 )

(isEq x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (join (equalL (! v1 L) (! v2 L))
                      (equalC (! v1 C) (! v2 C))
                      (equalZ (! v1 Z) (! v2 Z))))))
 (POST s2 v)
 )
```

```
(isNot x)
(begin
 (let post (Eval (NthSubexpr x 1) s))
 (let s1 (! post 1))
 (let v1 (! post 2))
 (let z (! v1 Z))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (notT (ZtoT z)))))
 (POST s1 v)
 )

(isOr x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (orT (ZtoT (! v1 Z)) (ZtoT (! v2 Z))))))
 (POST s2 v)
 )

(isAnd x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (andT (ZtoT (! v1 Z)) (ZtoT (! v2 Z))))))
 (POST s2 v)
 )

(isEtcBiOp x)
(begin
 (let post1 (Eval (NthSubexpr x 1) s))
 (let s1 (! post1 S))
 (let post2 (Eval (NthSubexpr x 2) s1))
 (let s2  (! post2 1))
 (let v (V (L bottom) (C bottom) (Z top)))        ;top integer value
 (POST s2 v)
 )

(isEtcUnOp x)
(begin
 (let post (Eval (NthSubexpr x 1) s))
 (let s1 (! post 1))
 (let v (V (L bottom) (C bottom) (Z top))) ;top integer value
 (POST s1 v)
 )

default   ; return top integer value for syscall
```

```
  (POST s (V (L bottom) (C bottom) (Z top)))
  )
 )
```

Appendix C

Specification for Eval2

```
;
;  instance-sensitive location
;
(types
 (domain PRE (* E S P))
 (domain POST (* S V))
 (domain S (-> Id SS))
 (domain SS (-> G V))
 (domain V (* L C Z))
 (domain L (* Ids P))
 (domain C (* Procs E))
 (domain P (-> Proc Gs))
 (domain E (synonym P))
 (domain Ids (2^ setId))
 (domain Id (flat setId))
 (domain Procs (2^ setProc))
 (domain Gs (2^ setG))
 (domain G (flat setG))
 (domain Z (2^ setZ))       (project Z (height> 2))
 (domain T (flat setB))
 (domain Proc (flat setProc))
 (set setG (elements EPS D DD))
 (set setId (index NumOfPlaces))
 (set setProc (index NumOfProcs))
 (set setB (elements True False))
 (set setZ (range -999999 999999))        ;-infinity to +infinity
)

; parser interface declarations
;(ExprSize (from syntree) (to setY))
;(PlaceSize (from syntree) (to setY))
(NumOfExprs NumOfProcs NumOfPlaces (from) (to number))
(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))
;(NthParamSize (from setProc number) (to setY))
;(NthLocalSize (from setProc number) (to setY))
;(LSize CSize ZSize (from) (to setY))
```

```
(Owner (from setId) (to setProc))
(DynamicId (from syntree) (to setId))
(ProcId (from syntree) (to setProc))
(Xid (from syntree) (to setId))
(Xnum (from syntree) (to number))
(Body (from setProc) (to syntree))
(isBegin isIf isRead isWrite isCreate isCall isApply isProc isClosure
 isId isNumber isPlus isMinus isMulti isMod isDiv
 isEq isLe isLt isGe isGt isNot isAnd isOr isEtcBiOp isEtcUnOp
 (from syntree) (to number))

;
; auxiliary semantic functions
;
(numToZ (from number) (to Z))
(function numToZ (n)
          (if (or (< n -99999) (> n 99999)) (Z top)
            (Z (setZ n))))

(plusZ (from Z Z) (to Z))
(function plusZ (z1 z2)
          (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
            (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
              (mapjoin (lambda (from setZ) (to Z) (setz1)
                         (mapjoin (lambda (from setZ) (to Z) (setz2)
                                    (numToZ (+ (number setz1)
                                                (number setz2))))
                                  z2))
                       z1))))

(minusZ (from Z Z) (to Z))
(function minusZ (z1 z2)
         (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
           (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
             (mapjoin (lambda (from setZ) (to Z) (setz1)
                        (mapjoin (lambda (from setZ) (to Z) (setz2)
                                   (numToZ (- (number setz1)
                                               (number setz2))))
                                 z2))
                      z1))))

(multiZ (from Z Z) (to Z))
(function multiZ (z1 z2)
          (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
            (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
              (mapjoin (lambda (from setZ) (to Z) (setz1)
                         (mapjoin (lambda (from setZ) (to Z) (setz2)
                                    (numToZ (* (number setz1)
                                                (number setz2))))
                                  z2))
                       z1))))

(numTosetB (from number) (to setB))
(function  numTosetB (n)  (if (= n 0) False True))

(ZtoT (from Z) (to T))
(function ZtoT (z)
          (if (= z (Z bottom)) (T bottom)
            (if (= z (Z top)) (T top)
              (if (<= z (Z (setZ 0)))        ; is z the singleton set {0}
```

```
                 (T False)
               ; z is a set of setZ elements
               (if (in (setZ 0) z) (T top) (T True))))))

(TtoZ (from T) (to Z))
(function TtoZ (t)
          (if (= t (T bottom)) (Z bottom)
            (if (= t (T top)) (Z top)
              (if (= t (T False)) (Z (setZ 0))
                (Z (setZ 1))))))

; abstract integer equal
(equalZ (from Z Z) (to T))
(function equalZ ( z1 z2)
      (if (or (= z1 (Z bottom))
              (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top))
                (= z2 (Z top))) (T top)
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (= (number setz1)
                                         (number setz2)))))
                                z2))
                     z1)))))

; abstract integer less than equal
(leZ (from Z Z) (to T))
(function leZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (<= (number setz1)
                                          (number setz2)))))
                                z2))
                     z1)))))

; abstract integer less than
(ltZ (from Z Z) (to T))
(function ltZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T False)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (< (number setz1)
                                         (number setz2)))))
                                z2))
                     z1)))))

; abstract integer greater than equal
(geZ (from Z Z) (to T))
(function geZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
```

```
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (<= (number setz2)
                                          (number setz1)))))
                                z2))
                     z1)))))

; abstract integer greater than
(gtZ (from Z Z) (to T))
(function gtZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T False)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (> (number setz1)
                                         (number setz2)))))
                                z2))
                     z1)))))

; abstract bool and
; note that when first is false, return false
(andT (from T T) (to T))
(function andT (t1 t2)
          (if (= t1 (T False)) t1
            (if (= t1 (T True)) t2
              (if (= t1 (T bottom)) t1
                (T top)))))

; abstract bool or
; note that when first is true, return true
(orT (from T T) (to T))
(function orT (t1 t2)
          (if (= t1 (T True)) t1
            (if (= t1 (T False)) t2
              (if (= t1 (T bottom)) t1
                (T top)))))

; abstract bool not
(notT (from T) (to T))
(function notT (t)
          (if (= t (T bottom)) (T bottom)
            (if (= t (T top)) (T top)
              (if (= t (T True)) (T False)
                (T True)))))

(equalL (from L L) (to T))
(function equalL (l1 l2)
          (if (or (= l1 (L bottom)) (= l2 (L bottom))) (T bottom)
            (if (or (= l1 (L top)) (= l2 (L top))) (T top)
              (if (= l1 l2) (T True) (T False)))))
(equalC (from C C) (to T))
(function equalC (c1 c2)
          (if (or (= c1 (C bottom)) (= c2 (C bottom))) (T bottom)
            (if (or (= c1 (C top)) (= c2 (C top))) (T top)
              (if (= c1 c2) (T True) (T False)))))
```

```
(instanceP (from setId P) (to Gs))
(function instanceP (id p)
          (@ p (Proc (Owner id))))

(circleS (from S setId Gs) (to V))
(function circleS (s id gs)
          (mapjoin (lambda (from setG) (to V) (g)
                     (@ (@ s (Id id)) (G g)))
                   gs))

(tfunction zeroGs (from) (to Gs)
           ( (Gs EPS))
           )
(tfunction zeroP (from) (to P)
           ( (P (each proc (zeroGs))))
           )

(tfunction moveG (from setG) (to Gs)
            (EPS  (Gs D))
            (D    (Gs DD))
            (DD   (Gs DD))
            )

(moveGs (from Gs) (to Gs))
(mfunction moveGs (gs)
           (mapjoin (lambda (from setG) (to Gs) (g)
                      (moveG g))
                    gs))

(moveP (from P setProc) (to P))
(mfunction moveP (p sproc)
           (begin
            (let proc (Proc sproc))
            ([/] p (by proc
                       (moveGs (@ p proc)))
                 )))

(slashSS (from SS V Gs) (to SS))
(mfunction slashSS (ss v gs)
           (join ss
                 (map[/] ss
                         (lambda (from setG) (to V) (g) v)
                         gs)))

;
; shadow e by the entries of P over the Ids
;
(OverShadow (from E P Ids Ids) (to E))
(mfunction OverShadow (e p params locals)
           (begin
            (let ids :Ids (join params locals))
            (let owners :Procs (join (for i 1 (|| ids)
                                          (Procs
                                            (Owner (nth ids i))))
                                     ))
            (map[/] e
                    (lambda (from setProc) (to Gs) (f)
                      (@ p (Proc f)))
                    owners))
           )
```

```
;
; The semantics function
;
; Note that PRE is the product of E, S, and P

(Eval (from syntree E S P) (to POST))        (start (zeroP) (S bottom) (zeroP))
(semantics Eval (x e s p)
  (switch
   (isBegin x)
   (begin
    (let (_post 1) (Eval (NthSubexpr x 1) e s p)
         (_post i) (for i 2 (NumOfSub x)
                        (Eval (NthSubexpr x i)
                              e
                              (! (_post (- i 1)) S)
                              p
                              )))
    (_post (NumOfSub x))
    )

   (isIf x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let s1 (! post1 S))
    (let v1 (! post1 V))
    (let cond (equalZ (! v1 Z) (Z (setZ 0))))
    (let ppost2 (if (<= (T False) cond)
                    (Eval (NthSubexpr x 2) e s1 p)
                  (POST bottom)))
    (let ppost3 (if (<= (T True) cond)
                    (Eval (NthSubexpr x 3) e s1 p)
                  (POST bottom)))
    (join ppost2 ppost3)
    )

   (isRead x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let s1 (! post1 S))
    (let v1 (! post1 V))
    (let l (! v1 L))
    (let ids (! l Ids))
    (let u (! l P))
    (let v2 (mapjoin (lambda (from setId) (to V) (i)
                       (circleS s1 i (instanceP i u)))
                     ids))
    (POST s1 v2)
    )

   (isWrite x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let v1 (! post1 V))
    (let post2 (Eval (NthSubexpr x 2) e (! post1 S) p))
    (let s2 (! post2 S))
    (let v2 (! post2 V))
    (let l (! v1 L))
    (let ids (! l Ids))
    (let u (! l P))
    (let s3 (mapjoin
```

```
         (lambda (from setId) (to S) (id)
           ([/] s2
                (by (Id id)
                    (slashSS (@ s2 (Id id)) v2 (instanceP id u)))))
         ids))
 (POST s3 v2)
 )

(isCreate x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1  (! post1 1))
 (let id  (DynamicId x))
 (let ss  (@ s1 (Id id)))
 (let l (L (Ids id)
          ([/] (P bottom) (by (Proc (Owner id))
                              (instanceP id e)))))
 (let v (V l (C bottom) (Z bottom)))
 (POST s1 v)
 )

(or (isProc x) (isClosure x))
(POST s (V (L bottom)
          (C (Procs (ProcId x)) e)
          (Z bottom)))

(isId x)
(begin
 (let id (Xid x))
 (let l (L (Ids id)
          ([/] (P bottom)
               (by (Proc (Owner id))
                   (instanceP id e)))))
 (POST s (V l (C bottom) (Z bottom)))
 )

(isNumber x)
(POST s (V (L bottom) (C bottom) (numToZ (Xnum x))))

(or (isCall x) (isApply x))
(begin
 (let n (NumOfSub x))
 (let num_actual (- n 1))
 (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 s1 (! (_post n) S))
 (let c (! (_v 1) C))
 (let procs (! c 1))
 (let f (! c 2))
 (mapjoin
  (lambda (from setProc) (to POST) (alpha)
     (begin
     (let mom (Body alpha))
     (let env1 (moveP f alpha))
     (let p1 (moveP p alpha))
```

```
        (let num_param (NumOfParam alpha))
        (let num_local (NumOfLocal alpha))
        (if (< num_actual  num_param)
            (POST bottom)
          (begin
           (let params :Ids (join (for i 1 num_param
                                       (Ids (NthParam alpha i)))))
           (let locals :Ids (join (for i 1 num_local
                                       (Ids (NthLocal alpha i)))))
           ;
           ; overshadow env1 with the entries of p1 for the locals and params
           ;
           (let env2 (OverShadow env1 p1 params locals))
           (let (_s2 0) s1        ; param binding
                (_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))
                                               (@ env2 (Proc alpha))))
                              ([/] (_s2 (- i 1)) (by (Id param) ss))))
                )
           ; initialize locals
;           (let (_s3 0) (_s2 num_param)
;                (_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))) ;initial local value
;                                       (@ env2 (Proc alpha))))
;                              ([/] (_s3 (- i 1)) (by (Id local) ss))))
;                )
           (Eval mom env2 (_s2 num_param) p1)
           ))
        ))
       procs
       ))

   (isPlus x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let s1 (! post1 1))
    (let v1 (! post1 2))
    (let post2 (Eval (NthSubexpr x 2) e s1 p))
    (let s2  (! post2 1))
    (let v2  (! post2 2))
    (let v (V (join (! v1 L) (! v2 L))
              (C bottom)
              (plusZ (! v1 Z) (! v2 Z))))
    (POST s2 v)
    )

   (isMulti x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let s1 (! post1 1))
    (let v1 (! post1 2))
```

```
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (multiZ (! v1 Z) (! v2 Z))))
 (POST s2 v)
 )

(isMinus x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (minusZ (! v1 Z) (! v2 Z))))
 (POST s2 v)
 )

(isLt x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (ltZ (! v1 Z) (! v2 Z)))))
 (POST s2 v)
 )

(isLe x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (leZ (! v1 Z) (! v2 Z)))))
 (POST s2 v)
 )

(isGe x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
```

```
          (C bottom)
          (TtoZ (geZ (! v1 Z) (! v2 Z)))))
 (POST s2 v)
 )

(isGt x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (ltZ (! v2 Z) (! v1 Z)))))
 (POST s2 v)
 )

(isEq x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (join (equalL (! v1 L) (! v2 L))
                      (equalC (! v1 C) (! v2 C))
                      (equalZ (! v1 Z) (! v2 Z))))))
 (POST s2 v)
 )

(isNot x)
(begin
 (let post (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post 1))
 (let v1 (! post 2))
 (let z (! v1 Z))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (notT (ZtoT z)))))
 (POST s1 v)
 )

(isOr x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let v1 (! post1 2))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let v2  (! post2 2))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ (orT (ZtoT (! v1 Z)) (ZtoT (! v2 Z))))))
 (POST s2 v)
 )
```

```
 (isAnd x)
 (begin
  (let post1 (Eval (NthSubexpr x 1) e s p))
  (let s1 (! post1 1))
  (let v1 (! post1 2))
  (let post2 (Eval (NthSubexpr x 2) e s1 p))
  (let s2  (! post2 1))
  (let v2  (! post2 2))
  (let v (V (L bottom)
           (C bottom)
           (TtoZ (andT (ZtoT (! v1 Z)) (ZtoT (! v2 Z))))))
  (POST s2 v)
  )

 (isEtcBiOp x)
 (begin
  (let post1 (Eval (NthSubexpr x 1) e s p))
  (let s1 (! post1 S))
  (let post2 (Eval (NthSubexpr x 2) e s1 p))
  (let s2  (! post2 1))
  (let v (V (L bottom) (C bottom) (Z top)))        ;top integer value
  (POST s2 v)
  )

 (isEtcUnOp x)
 (begin
  (let post (Eval (NthSubexpr x 1) e s p))
  (let s1 (! post 1))
  (let v (V (L bottom) (C bottom) (Z top))) ;top integer value
  (POST s1 v)
  )

 default   ; return top integer value for syscall
 (POST s (V (L bottom) (C bottom) (Z top)))
 )
)
```

Appendix D

Specification for Eval3

```
;
; instance-sensitive and block location
;
(types
 (domain PRE (* E S P))
 (domain POST (* S B))
 (domain S (-> Id SS))
 (domain SS (-> G B))
 (domain B (-> I V))
 (domain V (* L C Z))
 (domain L (* Ids P O))
 (domain C (* Procs E))
 (domain P (-> Proc Gs))
 (domain E (synonym P))
 (domain Ids (2^ setId))
 (domain Id (flat setId))
 (domain Procs (2^ setProc))
 (domain Gs (2^ setG))      ;(project Gs (height> 0))
 (domain G (flat setG))
 (domain Z (2^ setZ))       (project Z (height> 1))
 (domain O (flat setO))
 (domain T (flat setB))
 (domain Is (2^ setI))
 (domain I (flat setI))
 (domain Y (flat setY))
 (domain Proc (flat setProc))
 (set setI (* setO setY))               ; offset, size pair
 (set setG (elements EPS D DD))
 (set setId (index NumOfPlaces))
 (set setProc (index NumOfProcs))
 (set setB (elements True False))
 (set setY (range 1 16))               ; size 1 to 16 bytes
 (set setO (range 0 40))               ; offset
 (set setZ (range -999999 999999))       ;-infinity to +infinity
)

; parser interface declarations
(ExprSize (from syntree) (to setY))
(PlaceSize (from syntree) (to setY))
(NumOfExprs NumOfProcs NumOfPlaces (from) (to number))
```

```
(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))
(NthParamSize (from setProc number) (to setY))
(NthLocalSize (from setProc number) (to setY))
(LSize CSize ZSize (from) (to setY))
(Owner (from setId) (to setProc))
(DynamicId (from syntree) (to setId))
(ProcId (from syntree) (to setProc))
(Xid (from syntree) (to setId))
(Xnum (from syntree) (to number))
(Body (from setProc) (to syntree))
(isBegin isIf isRead isWrite isCreate isCall isApply isProc isClosure
 isId isNumber isPlus isMinus isMulti isMod isDiv
 isEq isLe isLt isGe isGt isNot isAnd isOr isEtcBiOp isEtcUnOp
 (from syntree) (to number))

;
; auxiliary semantic functions
;
(numToZ (from number) (to Z))
(function numToZ (n)
          (if (or (< n -99999) (> n 99999)) (Z top)
            (Z (setZ n))))
(numToO (from number) (to O))
(function numToO (n)
          (if (or (< n 0) (> n 400)) (O top)
            (O (setO n))))

(offsetSize (from setO setY) (to I))
(function offsetSize (z y) (I (setI z y)))

(plusOZ (from O Z) (to O))
(function plusOZ (o z)
            (if (or (= o (O bottom)) (= z (Z bottom))) (O bottom)
              (if (or (= o (O top)) (= z (Z top))) (O top)
                (mapjoin
                 (lambda (from setZ) (to O) (setz)
                   (numToO (+ (number setz) (number (setO o)))))
                 z))))

(minusOZ (from O Z) (to O))
(function minusOZ (o z)
            (if (or (= o (O bottom)) (= z (Z bottom))) (O bottom)
              (if (or (= o (O top)) (= z (Z top))) (O top)
                (mapjoin
                 (lambda (from setZ) (to O) (setz)
                   (numToO (- (number (setO o)) (number setz))))
                 z))))

(plusZ (from Z Z) (to Z))
(function plusZ (z1 z2)
          (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
            (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
              (mapjoin (lambda (from setZ) (to Z) (setz1)
                         (mapjoin (lambda (from setZ) (to Z) (setz2)
                                    (numToZ (+ (number setz1)
                                               (number setz2))))
```

```
                                z2))
                      z1))))

(minusZ (from Z Z) (to Z))
(function minusZ (z1 z2)
         (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
           (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
             (mapjoin (lambda (from setZ) (to Z) (setz1)
                        (mapjoin (lambda (from setZ) (to Z) (setz2)
                                   (numToZ (- (number setz1)
                                               (number setz2))))
                                 z2))
                      z1))))

(multiZ (from Z Z) (to Z))
(function multiZ (z1 z2)
          (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (Z bottom)
            (if (or (= z1 (Z top)) (= z2 (Z top))) (Z top)
              (mapjoin (lambda (from setZ) (to Z) (setz1)
                         (mapjoin (lambda (from setZ) (to Z) (setz2)
                                    (numToZ (* (number setz1)
                                                (number setz2))))
                                  z2))
                       z1))))

(numTosetB (from number) (to setB))
(function  numTosetB (n)  (if (= n 0) False True))

(ZtoT (from Z) (to T))
(function ZtoT (z)
          (if (= z (Z bottom)) (T bottom)
            (if (= z (Z top)) (T top)
              (if (<= z (Z (setZ 0)))        ; is z the singleton set {0}
                  (T False)
                ; z is a set of setZ elements
                (if (in (setZ 0) z) (T top) (T True))))))

(TtoZ (from T) (to Z))
(function TtoZ (t)
          (if (= t (T bottom)) (Z bottom)
            (if (= t (T top)) (Z top)
              (if (= t (T False)) (Z (setZ 0))
                (Z (setZ 1))))))

; abstract integer equal
(equalZ (from Z Z) (to T))
(function equalZ ( z1 z2)
      (if (or (= z1 (Z bottom))
              (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top))
                (= z2 (Z top))) (T top)
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (= (number setz1)
                                         (number setz2)))))
                                z2))
                     z1)))))
```

```
; abstract integer less than equal
(leZ (from Z Z) (to T))
(function leZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (<= (number setz1)
                                          (number setz2)))))
                                z2))
                     z1)))))

; abstract integer less than
(ltZ (from Z Z) (to T))
(function ltZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T False)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (< (number setz1)
                                         (number setz2)))))
                                z2))
                     z1)))))

; abstract integer greater than equal
(geZ (from Z Z) (to T))
(function geZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T True)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (<= (number setz2)
                                          (number setz1)))))
                                z2))
                     z1)))))

; abstract integer greater than
(gtZ (from Z Z) (to T))
(function gtZ (z1 z2)
      (if (or (= z1 (Z bottom)) (= z2 (Z bottom))) (T bottom)
        (if (or (= z1 (Z top)) (= z2 (Z top))) (T top)
          (if (= z1 z2) (T False)
            (mapjoin (lambda (from setZ) (to T) (setz1)
                       (mapjoin (lambda (from setZ) (to T) (setz2)
                                  (T (numTosetB
                                      (> (number setz1)
                                         (number setz2)))))
                                z2))
                     z1)))))

; abstract bool and
; note that when first is false, return false
(andT (from T T) (to T))
(function andT (t1 t2)
```

```
          (if (= t1 (T False)) t1
            (if (= t1 (T True)) t2
              (if (= t1 (T bottom)) t1
                (T top)))))

; abstract bool or
; note that when first is true, return true
(orT (from T T) (to T))
(function orT (t1 t2)
          (if (= t1 (T True)) t1
            (if (= t1 (T False)) t2
              (if (= t1 (T bottom)) t1
                (T top)))))

; abstract bool not
(notT (from T) (to T))
(function notT (t)
          (if (= t (T bottom)) (T bottom)
            (if (= t (T top)) (T top)
              (if (= t (T True)) (T False)
                (T True)))))

(equalL (from L L) (to T))
(function equalL (l1 l2)
          (if (or (= l1 (L bottom)) (= l2 (L bottom))) (T bottom)
            (if (or (= l1 (L top)) (= l2 (L top))) (T top)
              (if (= l1 l2) (T True) (T False)))))
(equalC (from C C) (to T))
(function equalC (c1 c2)
          (if (or (= c1 (C bottom)) (= c2 (C bottom))) (T bottom)
            (if (or (= c1 (C top)) (= c2 (C top))) (T top)
              (if (= c1 c2) (T True) (T False)))))

(instanceP (from setId P) (to Gs))
(function instanceP (id p)
          (@ p (Proc (Owner id))))

(circleS (from S setId Gs) (to B))
(function circleS (s id gs)
          (mapjoin (lambda (from setG) (to B) (g)
                     (@ (@ s (Id id)) (G g)))
                   gs))

(tfunction zeroGs (from) (to Gs)
           ( (Gs EPS))
           )
(tfunction zeroP (from) (to P)
           ( (P (each proc (zeroGs))))
           )

(restrictB (from B Is) (to B))
(mfunction restrictB (b is)
           (if (= is (Is top)) b
             (map[/] (B bottom)
                     (lambda (from setI) (to V) (i) (@ b (I i)))
                     is)))

(ReadIs (from Is O setY) (to Is))
(mfunction ReadIs (is o sety)        ; note that arguments are all non-tops.
           (mapjoin (lambda (from setI) (to Is) (i)
```

```
                    (begin
                     (let z (number (setO o)))
                     (let k (number (setZ sety)))
                     (let zp (number (! i 1)))
                     (let kp (number (setZ (! i 2))))
                     (if (and (<= z zp) (<= (+ zp kp) (+ z k)))
                         (Is i)
                       (Is bottom))))
                  is)
          )

(Read (from B O setY) (to B))
(mfunction Read (b z sety)
          (if (or (= z (O bottom)) (= b (B bottom))) (B bottom)
            (if (= z (O top)) b
              (if (= b (B top))
                  ([/] (B bottom) (by (I (setI (setO O) sety))
                                      (V top)))
                (begin
                 (let k (number (setZ sety)))
                 (let is (dom b))
                 (let iss (ReadIs is z sety))
                 (map[/] (B bottom)
                         (lambda (from setI) (to V) (i)
                           (@ b (I i)))
                         (lambda (from setI) (to I) (i)
                           (begin
                            (let zp (! i setO))
                            (let kp (! i setY))
                            (I (setI (setO (- (number zp)
                                              (number (setZ (setO z)))))
                                     kp))))
                         iss)
                 )))))

(WriteIsUc (from Is O setY) (to Is))
(mfunction WriteIsUc (is o sety)
          (mapjoin (lambda (from setI) (to Is) (i)
                    (begin
                     (let z (number (setO o)))
                     (let k (number (setZ sety)))
                     (let zp (number (! i 1)))
                     (let kp (number (setZ (! i 2))))
                     (if (and (<= z zp) (<= (+ zp kp) (+ z k)))
                         (Is i)
                       (Is bottom))))
                  is)
          )

(WriteIsNc (from Is O setY) (to Is))
(mfunction WriteIsNc (is o sety)
     (mapjoin (lambda (from setI) (to Is) (i)
                (begin
                 (let z (number (setO o)))
                 (let k (number (setZ sety)))
                 (let zp (number (! i 1)))
                 (let kp (number (setZ (! i 2))))
                 (if (and (<= z zp) (<= (+ zp kp) (+ z k)))
                     (Is i)
                   (Is bottom))))
```

```
             is)
     )

(WriteIsNo (from Is O setY) (to Is))
(mfunction WriteIsNo (is o sety)
     (mapjoin (lambda (from setI) (to Is) (i)
                (begin
                 (let zp (number (! i 1)))
                 (let kp (number (setZ (! i 2))))
                 (let z (number (setO o)))
                 (let k (number (setZ sety)))
                 (if (and (< z (+ zp kp)) (< zp (+ z k)))
                     (Is i)
                   (Is bottom))))
             is)
     )

(WriteBsh (from B O) (to B))
(mfunction WriteBsh (b o)                ;o should not be (O top)
           (if (= b (B top)) b
             (begin
              (let is (dom b))
              (map[/] (B bottom)
                      (lambda (from setI) (to V) (i) (@ b (I i)))
                      (lambda (from setI) (to I) (i)
                        (begin
                         (let zp (! i setO))
                         (let kp (! i setY))
                         (I (setI (setO (+ (number (setO o)) (number zp)))
                                  kp))))
                      is)
              )))

(Write (from B B O setY) (to B))
(mfunction Write (bd bs o sety)
           (switch
            (or (= bs (B bottom)) (= o (O bottom)))
            (B bottom)

            (or (and (= bs (B top)) (= o (O top)))
                (and (= bd (B top)) (= o (O top)))
                (and (= bd (B top)) (= bs (B top))))
            (B top)

            (= o (O top))                ; only z is top
            (B top)                        ; \lambda<z,k>.bs(O,k)
                                        ; no way to express the above
            (= bd (B top))                ;only bd is top
            (B top)

            (= bs (B top))                ;only bs is top, bd can be bottom
            (begin
             (let bs_p ([/] (B bottom)
                        (by (offsetSize (setO 0) sety) (V top))))
             (let bsh (WriteBsh bs_p o))
             (let dombd (dom bd))
             (let isUc :Is (WriteIsUc dombd o sety))
             (let isNc :Is (WriteIsNc (dom bsh) o sety))
             (let isNo :Is (WriteIsNo dombd o sety))
             (let b1 (join bd (restrictB bsh isUc)))
```

```
             (let b2 (map[/] b1
                             (lambda (from setI) (to V) (i) (V bottom))
                             isNo))
             (map[/] b2
                     (lambda (from setI) (to V) (i) (@ bsh (I i)))
                     isNc)
             )

            default                        ;bd, bs, z are not tops
            (begin
             (let bsh (WriteBsh bs o))
             (let dombd (dom bd))
             (let isUc :Is (WriteIsUc dombd o sety))
             (let isNc :Is (WriteIsNc (dom bsh) o sety))
             (let isNo :Is (WriteIsNo dombd o sety))
             (let b1 (join bd (restrictB bsh isUc)))
             (let b2 (map[/] b1
                             (lambda (from setI) (to V) (i) (V bottom))
                             isNo))
             (map[/] b2
                     (lambda (from setI) (to V) (i) (@ bsh (I i)))
                     isNc)
             )))

(tfunction moveG (from setG) (to Gs)
            (EPS  (Gs D))
            (D    (Gs DD))
            (DD   (Gs DD))
            )

(moveGs (from Gs) (to Gs))
(mfunction moveGs (gs)
           (mapjoin (lambda (from setG) (to Gs) (g)
                      (moveG g))
                   gs))

(moveP (from P setProc) (to P))
(mfunction moveP (p sproc)
           (begin
            (let proc (Proc sproc))
            ([/] p (by proc
                       (moveGs (@ p proc)))
                 )))

(slashSS (from SS B Gs) (to SS))
(mfunction slashSS (ss b gs)
           (join ss
                 (map[/] ss
                         (lambda (from setG) (to B) (g) b)
                         gs)))

;
; shadow e by the entries of P over the Ids
;
(OverShadow (from E P Ids Ids) (to E))
(mfunction OverShadow (e p params locals)
           (begin
            (let ids :Ids (join params locals))
            (let owners :Procs (join (for i 1 (|| ids)
                                          (Procs
```

```
                                        (Owner (nth ids i))))
                                    ))
            (map[/] e
                    (lambda (from setProc) (to Gs) (f)
                      (@ p (Proc f)))
                    owners))
           )

(initLocal (from setY) (to B))
(mfunction initLocal (local_size)
           ([/] (B bottom)
                (by (offsetSize (set0 0) local_size)
                    (V (L bottom) (C bottom) (Z (setZ 0))))))

; The semantics function
;
(Eval (from syntree E S P) (to POST))       (start (zeroP) (S bottom) (zeroP))
(semantics Eval (x e s p)
  (switch
   (isBegin x)
   (begin
    (let (_post 1) (Eval (NthSubexpr x 1) e s p)
         (_post i) (for i 2 (NumOfSub x)
                      (Eval (NthSubexpr x i)
                            e
                            (! (_post (- i 1)) S)
                            p)))
    (_post (NumOfSub x))
    )

   (isIf x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let s1 (! post1 S))
    (let b1 (! post1 B))
    (let cond (equalZ (! (@ b1 (offsetSize (set0 0)
                                       (ExprSize (NthSubexpr x 1))))
                         Z)
                      (Z (setZ 0))))
    (let ppost2 (if (<= (T False) cond)
                    (Eval (NthSubexpr x 2) e s1 p)
                  (POST bottom)))
    (let ppost3 (if (<= (T True) cond)
                    (Eval (NthSubexpr x 3) e s1 p)
                  (POST bottom)))
    (join ppost2 ppost3)
    )

   (isRead x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let s1 (! post1 1))
    (let b1 (! post1 2))
    (let l (! (@ b1 (offsetSize (set0 0) (LSize))) L))
    (let ids (! l 1))
    (let u (! l 2))
    (let z (! l 3))
    (let b2 (mapjoin (lambda (from setId) (to B) (i)
                       (Read (circleS s1 i (instanceP i u))
                             z
```

```
                         (setY (ExprSize x))))
                 ids))
   (POST s1 b2)
   )

  (isWrite x)
  (begin
   (let post1 (Eval (NthSubexpr x 1) e s p))
   (let b1 (! post1 2))
   (let post2 (Eval (NthSubexpr x 2) e (! post1 1) p))
   (let s2 (! post2 1))
   (let b2 (! post2 2))
   (let l (! (@ b1 (offsetSize (set0 0) (LSize))) L))
   (let ids (! l 1))
   (let u (! l 2))
   (let z (! l 3))
   (let k :setY (ExprSize (NthSubexpr x 2)))
   (let s3 (mapjoin
            (lambda (from setId) (to S) (id)
              ([/] s2
                   (by (Id id)
                       (slashSS (@ s2 (Id id))
                                (Write
                                 (circleS s2 id (instanceP id u))
                                 b2
                                 z
                                 k)
                                (instanceP id u)))))
            ids))
   (POST s3 b2)
   )

  (isCreate x)
  (begin
   (let post1 (Eval (NthSubexpr x 1) e s p))
   (let s1  (! post1 1))
   (let id  (DynamicId x))
   (let ss  (@ s1 (Id id)))
   (let l (L (Ids id)
             ([/] (P bottom) (by (Proc (Owner id))
                                 (instanceP id e)))
             (O (set0 0))))
   (let v (V l (C bottom) (Z bottom)))
   (let b2 ([/] (B bottom) (by (offsetSize (set0 0) (LSize)) v)))
   (POST s1 b2)
   )

  (or (isProc x) (isClosure x))
  (begin
   (let c (C (Procs (ProcId x)) e))
   (let b ([/] (B bottom)
               (by (offsetSize (set0 0) (CSize))
                   (V (L bottom) c (Z bottom)))))
   (POST s b))

  (isId x)
  (begin
   (let id (Xid x))
   (let l (L (Ids id)
             ([/] (P bottom)
```

```
                (by (Proc (Owner id))
                    (instanceP id e)))
           (O (setO 0))))
 (let b ([/] (B bottom)
               (by (offsetSize (setO 0) (LSize))
                   (V 1 (C bottom) (Z bottom)))))
 (POST s b)
 )

(isNumber x)
(begin
 (let b ([/] (B bottom)
               (by (offsetSize (setO 0) (ExprSize x))
                   (V (L bottom) (C bottom) (numToZ (Xnum x))))))
 (POST s b)
 )

(or (isCall x) (isApply x))
(begin
 (let n (NumOfSub x))
 (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 (_b i) (for i 1 n (! (_post i) B)))
 (let s1 (! (_post n) S))
 (let c (! (@ (_b 1) (offsetSize (setO 0) (CSize))) C))
 (let procs (! c 1))
 (let f (! c 2))
 (mapjoin
  (lambda (from setProc) (to POST) (alpha)
    (begin
     (let mom (Body alpha))
     (let env1 (moveP f alpha))
     (let p1 (moveP p alpha))
     (let (_b1 1) (B bottom)
          (_b1 i) (for i 2 n
                       (Write (_b1 (- i 1)) (_b i)
                              (numToO (+ (for j 2 (- i 1)
                                                (number
                                                 (ExprSize
                                                  (NthSubexpr x j))))))
                              (setY (ExprSize (NthSubexpr x i))))))
     (let num_param (NumOfParam alpha))
     (let params :Ids (join (for i 1 num_param (Ids (NthParam alpha i)))))
     (let num_local (NumOfLocal alpha))
     (let locals :Ids (join (for i 1 num_local (Ids (NthLocal alpha i)))))
     (let env2 (OverShadow env1 p1 params locals))
     ; param binding
     (let (_s2 0) s1
          (_s2 i) (for i 1 num_param
                       (begin
                        (let param :setId (nth params i))
                        (let ss (slashSS (@ (_s2 (- i 1)) (Id param))
                                         (Read
                                          (_b1 n)
                                          (numToO
                                           (+ (for j 1 (- i 1)
```

```
                                                    (number
                                                     (NthParamSize
                                                      alpha j)))))
                                              (NthParamSize alpha i)
                                              )
                                             (@ env2 (Proc alpha))))
                         ([/] (_s2 (- i 1)) (by (Id param) ss))))
            )
        ; initialize local
;        (let (_s3 0) (_s2 num_param)
;             (_s3 i) (for i 1 num_local
;                          (begin
;                           (let local :setId (nth locals i))
;                           (let ss (slashSS
;                                    (@ (_s3 (- i 1)) (Id local))
;                                    (initLocal (NthLocalSize alpha i))
;                                    (@ env2 (Proc alpha))))
;                           ([/] (_s3 (- i 1)) (by (Id local) ss))))
;            )
        (let post2 (Eval mom env2 (_s2 num_param) p1))
        (let b3 (Read (! post2 2) (O (setO 0)) (ExprSize x)))
        (POST (! post2 S) b3)
        ))
     procs
     ))

   (isPlus x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let s1 (! post1 1))
    (let b1 (! post1 2))
    (let v1 (@ b1 (offsetSize (setO 0) (ExprSize (NthSubexpr x 1)))))
    (let post2 (Eval (NthSubexpr x 2) e s1 p))
    (let s2  (! post2 1))
    (let b2  (! post2 2))
    (let v2  (@ b2 (offsetSize (setO 0) (ExprSize (NthSubexpr x 2)))))
    (let l1  (! v1 1))
    (let l2  (! v2 1))
    (let z1  (! v1 3))
    (let z2  (! v2 3))
    (let l (L (join (! l1 1) (! l2 1))
              (join (! l1 2) (! l2 2))
              (join (plusOZ (! l1 0) z2) (plusOZ (! l2 0) z1))))
    (let b ([/] (B bottom)
                (by (offsetSize (setO 0) (ExprSize x))
                    (V 1 (C bottom) (plusZ z1 z2)))))
    (POST s2 b)
    )

   (isMulti x)
   (begin
    (let post1 (Eval (NthSubexpr x 1) e s p))
    (let s1 (! post1 1))
    (let b1 (! post1 2))
    (let v1 (@ b1 (offsetSize (setO 0) (ExprSize (NthSubexpr x 1)))))
    (let post2 (Eval (NthSubexpr x 2) e s1 p))
    (let s2 (! post2 1))
    (let b2 (! post2 2))
    (let v2 (@ b2 (offsetSize (setO 0) (ExprSize (NthSubexpr x 2)))))
    (let z1 (! v1 3))
```

```
 (let z2 (! v2 3))
 (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x))
                (V (L bottom) (C bottom) (multiZ z1 z2)))))
 (POST s2 b)
 )

(isMinus x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let b1 (! post1 2))
 (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let b2  (! post2 2))
 (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
 (let z1  (! v1 3))
 (let z2  (! v2 3))
 (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x))
                (V (L bottom) (C bottom) (minusZ z1 z2)))))
 (POST s2 b)
 )

(isLt x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let b1 (! post1 2))
 (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let b2  (! post2 2))
 (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
 (let t (ltZ (! v1 Z) (! v2 Z)))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ t)))
 (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x))
                v)))
 (POST s2 b)
 )

(isLe x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let b1 (! post1 2))
 (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let b2  (! post2 2))
 (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
 (let t (leZ (! v1 Z) (! v2 Z)))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ t)))
 (let b ([/] (B bottom)
```

```
            (by (offsetSize (set0 0) (ExprSize x))
                v)))
 (POST s2 b)
 )

(isGe x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let b1 (! post1 2))
 (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let b2  (! post2 2))
 (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
 (let t (geZ (! v1 Z) (! v2 Z)))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ t)))
 (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x))
                v)))
 (POST s2 b)
 )

(isGt x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let b1 (! post1 2))
 (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let b2  (! post2 2))
 (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
 (let t (gtZ (! v1 Z) (! v2 Z)))
 (let v (V (L bottom)
          (C bottom)
          (TtoZ t)))
 (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x))
                v)))
 (POST s2 b)
 )

(isEq x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1  (! post1 1))
 (let b1  (! post1 2))
 (let v1  (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let b2  (! post2 2))
 (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
 (let l1  (! v1 1))
 (let l2  (! v2 1))
 (let c1  (! v1 2))
 (let c2  (! v2 2))
 (let z1  (! v1 3))
```

```
 (let z2  (! v2 3))
 (let t (join (equalL l1 l2)
              (equalC c1 c2)
              (equalZ z1 z2)))
 (let v (V (L bottom)
           (C bottom)
           (TtoZ t)))
 (let b ([/] (B bottom)
             (by (offsetSize (set0 0) (ExprSize x))
                 v)))
 (POST s2 b)
 )

(isNot x)
(begin
 (let post (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post 1))
 (let b1 (! post 2))
 (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let z (! v1 Z))
 (let v (V (L bottom)
           (C bottom)
           (TtoZ (notT (ZtoT z)))))
 (let b ([/] (B bottom)
             (by (offsetSize (set0 0) (ExprSize x))
                 v)))
 (POST s1 b)
 )

(isOr x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let b1 (! post1 2))
 (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let b2  (! post2 2))
 (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
 (let z1 (! v1 Z))
 (let z2 (! v2 Z))
 (let v (V (L bottom)
           (C bottom)
           (TtoZ (orT (ZtoT z1) (ZtoT z2)))))
 (let b ([/] (B bottom)
             (by (offsetSize (set0 0) (ExprSize x))
                 v)))
 (POST s2 b)
 )

(isAnd x)
(begin
 (let post1 (Eval (NthSubexpr x 1) e s p))
 (let s1 (! post1 1))
 (let b1 (! post1 2))
 (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
 (let post2 (Eval (NthSubexpr x 2) e s1 p))
 (let s2  (! post2 1))
 (let b2  (! post2 2))
 (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
```

```
  (let z1 (! v1 Z))
  (let z2 (! v2 Z))
  (let v (V (L bottom)
          (C bottom)
          (TtoZ (andT (ZtoT z1) (ZtoT z2)))))
  (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x))
                v)))
  (POST s2 b)
  )

 (isEtcBiOp x)
 (begin
  (let post1 (Eval (NthSubexpr x 1) e s p))
  (let s1 (! post1 1))
  (let b1 (! post1 2))
  (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
  (let post2 (Eval (NthSubexpr x 2) e s1 p))
  (let s2  (! post2 1))
  (let b2  (! post2 2))
  (let v2  (@ b2 (offsetSize (set0 0) (ExprSize (NthSubexpr x 2)))))
  (let z1 (! v1 Z))
  (let z2 (! v2 Z))
  (let v (V (L bottom)
          (C bottom)
          (Z top)))                          ;top integer value
  (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x))
                v)))
  (POST s2 b)
  )

 (isEtcUnOp x)
 (begin
  (let post (Eval (NthSubexpr x 1) e s p))
  (let s1 (! post 1))
  (let b1 (! post 2))
  (let v1 (@ b1 (offsetSize (set0 0) (ExprSize (NthSubexpr x 1)))))
  (let z (! v1 Z))
  (let v (V (L bottom)
          (C bottom)
          (Z top)))
  (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x)) v)))
  (POST s1 b)
  )

 default   ; return top integer value for syscall
 (begin
  (let v (V (L bottom)
          (C bottom)
          (Z top)))
  (let b ([/] (B bottom)
            (by (offsetSize (set0 0) (ExprSize x)) v)))
  (POST s b)
  )
 )
)
```

VITA

Kwangkeun Yi was born in 1965, Korea. He entered into the College of Natural Science, Seoul National University in 1983. He majored in Computer Science and Statistics and graduated in 1987. In Fall 1987, he started his graduate studies in the University of Illinois at Urbana-Champaign. During his graduate years, he has been a research assistant in the Center for Supercomputing Research and Development. After completing his Ph.D. degree, he joined AT&T Bell Labs' Software Principles Research Laboratory at Murray Hill, New Jersey. His interest is in optimizing compilers, programming language semantics, and structural art.