A Machine Description Facility for Compiler Testing

HANAN SAMET, MEMBER, IEEE

Abstract—Requirements for a machine description facility for compiler testing are discussed. The compiler testing procedure consists of proving that programs are correctly translated by the compiler at hand. This is achieved by use of a common intermediate representation for both the source and object programs. The intermediate representation for the object program is built by use of a process termed symbolic interpretation. This process interprets a set of procedures which describe the effects of machine language instructions corresponding to the target machine on a suitable computation model in a manner consistent with an execution level definition of the high level language. Some of the important factors which enter into such a definition are discussed. These include architectural constraints posed by the target machine, and a description facility for memory and data types. Once such a definition is formulated, the actual instruction set of the target machine can be described. The highlights and limitations of such a definition facility are discussed in the context of a specific LISP implementation on a PDP-10 computer.

Index Terms—Compilers, compiler testing, correctness, data types, LISP, machine description languages, program testing, program verification.

I. Introduction

GIVEN A computer, one of the first questions that comes to mind is what are its capabilities. This question is often answered by describing its instruction set. This description process is generally geared towards a specific application. These applications include very low level wiring diagrams at the electronic gate level, microprograms [8], and higher level register transfer languages [1]. In this paper we present a formalism for describing a computer for compiler testing.

In order to motivate our ideas we define the concept of compiler testing and discuss its relationship to other work. Once this is done we present a machine description facility that is suited to compiler testing. The facility is decomposed into two parts—instructions and memory. The presentation draws heavily on examples from an existing compiler testing system [19].

II. Compiler Testing

Compiler testing is a term we use to describe a means of proving that given a compiler (or any program translation procedure) and a program to be compiled, the translation has been correctly performed. This concept is useful when the translator exhibits a considerable amount of optimization since it is not unusual for optimizations to result in erroneous program behavior. Some possible approaches to tackling this issue include program proving [14], program testing [7], and decompilation techniques [6].

Program proving methods have traditionally been characterized by the specification of assertions ([5], [10]) about the intent of the program and then proving that they do indeed hold. Such techniques have two drawbacks. First, the process of specifying assertions as to what constitutes correct program behavior is not easy ([2], [22]), and even when a program has been found to satisfy the supplied assertions there is no guarantee that the assertions are sufficiently precise to account for all contingencies. The assertions generally deal with what will be termed the intent or "high level" behavior of the program, whereas we are interested in "low level" behavior. Some notable work using assertions in verifying low level behavior is reported in [12] and [13]. Second, proofs using assertions rely on the existence of a theorem prover, and a correctness proof for a compiler can be characterized as proving that there does not exist a program that is incorrectly compiled. This is in contrast to an alternative program testing approach which would prove that specific programs are correctly translated on a case by case basis.

Decompilation methods could conceivably be used to verify the equivalence of a source program and an object program. This would require *a priori* knowledge of how the various constructs in the source language have been encoded in the object language. However, such an approach sets a limit on the variation in the object code that can be presented to such a system. A more serious flaw is the fact that compilation is a many-to-many process. Namely, the object program corresponding to a program written in a high level language can be encoded in many equivalent ways. Similarly, to an object program there corresponds more than one equivalent source program.

Our notion of compiler testing is a variation on the concept of program testing. We feel that in the case of a compiler there exists a willingness to settle for proofs that specific programs are correctly translated from a high level language to the object language. Thus a proof system is embedded in the compiler which proves the correctness of the translation for each program input to the compiler. This sidesteps the issue of proving that there does not exist a program that is incorrectly compiled; but this issue is now moot since essentially we are only interested in the correctness of translations of the programs input to the compiler. In other words we are not concerned with the correctness of translation of programs that have not been input to the compiler. Thus our variation enables us to bootstrap ourselves to a state where we can attribute an effective correctness to the compiler.

Our test criterion for compiler testing is a proof of equivalence between a program input to the compiler and the corresponding translated object program. The manner in which we proceed is to find an intermediate representation which is com-

Manuscript received April 23, 1976; revised September 20, 1976. This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435. The views expressed are those of the author.

The author is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

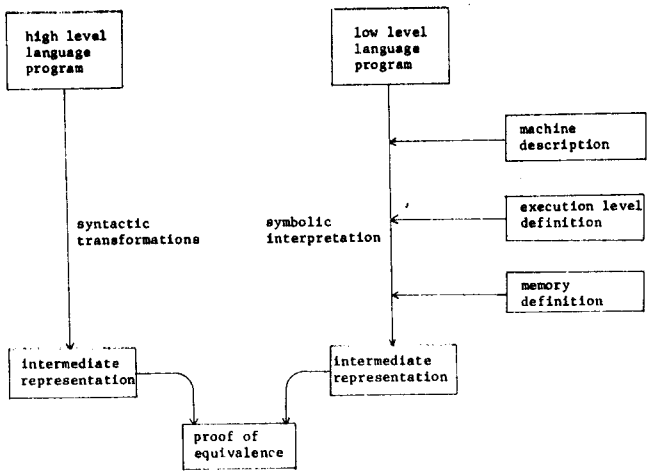

Fig. 1. Compiler testing system diagram.

mon to both the original and object programs and then check for equivalence. This relies on the existence of such a representation. In addition, we must be more precise in our definition of equivalence. By equivalence, we mean that the two programs must be capable of being proved to be structurally equivalent [11]—i.e., they have identical execution sequences except for certain valid rearrangements of computations. No use is made of the purpose of the program in the process of proving equivalence. Thus, for example, we can not prove that a high level sorting program using insertion sort is equivalent to a low level sorting program using quicksort since the notion of sorting is an input/output pair characterization of an algorithm.

The actual testing procedure consists of three steps (see Fig. 1). First, the high level language program must be converted to the intermediate representation. Second, the low level language program must be converted to the intermediate representation. Third, a check must be performed of the equivalence of the two representations. This check takes the form of a procedure which applies valid equivalence preserving transformations to the results of the first two steps in attempting to reduce them to a common representation.

The heart of the testing procedure is an intermediate representation common to both the source and object programs. Such a representation must be chosen with care if the problems alluded to in the discussion of decompilation techniques are to be avoided. Recall that we indicated that due to the many-to-many nature of the translation mapping in the case of optimization, we are hard pressed in obtaining the original representation of the program unless we give an *a priori* formulation of how all the constructs of the high level language are encoded in the low level language. Alternatively, if the intermediate representation were the object language itself, and the transformation process from the high level language to the object language were the process of compilation, then the process of proving equivalence would be trivial.

Clearly the intermediate representation and the process of obtaining it are very dependent on the high and low level languages. For example, in the case of LISP [15] such an intermediate representation has been shown to exist [20]. This representation is in the form of a tree with each nonterminal node denoting a predicate. In this case the representation is obtained by applying a valid set of syntactic transformations to the original source program. In general, the intermediate representation and the process of obtaining it must be capable of being proved correct, or believable, with greater ease than the compilation itself.

The intermediate representation in the second step of the testing procedure is built by a process termed symbolic interpretation. This process consists of activating a set of procedures corresponding to instructions in the low level program consistent with an execution level definition of the high level language (similar to interpretation). These procedures specify how each instruction effects an entity known as the computation model (e.g., procedural embedding [23]). This model reflects the contents of the various data structures relevant to the execution of the program, as well as the values of the conditions tested. Note that the validity of the process of converting the low level program to the intermediate representation is shown by proving that the instruction descriptions correctly update the computation model.

The symbolic interpretation procedure begins at the starting address of the low level program and updates the computation model according to the instruction's procedural description. Note that we are assuming that self-modifying code is prohibited. If an instruction corresponds to a test, then an attempt is made to determine if the value of the condition is already known. In the affirmative case, symbolic interpretation resumes at the next logical instruction. In the negative case, both alternatives of the condition are symbolically interpreted. In case the instruction corresponds to recursion or a call on an external function, then the function being called is not symbolically interpreted and control resumes at the next instruction which is to be executed upon the termination of the function call. In addition, when processing an instruction that has been previously encountered along the path being symbolically interpreted, recursion is assumed to have taken place, and the symbolic interpretation procedure attempts to prove that if recursion had indeed taken place, then the said instruction would have been reached with the same state of the computation model by virtue of known values for all of the conditions along some path to the said instruction. The manner in which external function calls and recursion are treated, as well as the prohibition of self-modifying code, guarantees the termination of the symbolic interpretation procedure.

We refer to the computation model and to the procedures corresponding to the instructions collectively as a machine description facility. In the following sections we describe the techniques necessary to provide a machine description facility for symbolic interpretation. These include a means of describing memory (i.e., the environment in which the program is being symbolically interpreted), a mechanism for keeping track of the data types associated with the various locations, and an operational definition of the various instructions of the object language. However, in order to have some framework for the discussion, we must assume the existence of a suitable programming language and an execution level definition

for the language. Our high level language is a subset of LISP 1.6 [18], a variant of LISP, which has been shown to have a suitable intermediate representation [20]. The low level language is LAP [18] (a variant of the PDP-10 [3] assembly language). An actual proof system employing the ideas discussed here is described in [19].

Section III contains a discussion of what constitutes an execution level definition. Section IV uses some of the ideas of Section III to give an example of compiler testing. The example consists of a high level language program, a low level language program, the corresponding intermediate representations, and a brief discussion of how their equivalence is demonstrated. Sections V and VI present the formalism used in the symbolic interpretation procedure to describe a computer instruction set and record its effect on a computation model.

III. Execution Level Definition

The execution level definition of a high level language must take several factors into account. A primary factor is the set of architectural constraints that are posed by the target computer. A second, and equally important factor, concerns conventions to which object level programs must adhere with respect to control structures of the high level language.

Architectural constraints involve inherent properties of the target computer and have a direct effect on the execution level definition. These constraints include word size, instruction format, type of arithmetic (two's complement or one's complement), addressing structure, existence of a hardware stack, memory management, overflow and underflow detection, etc. They have a direct effect on the low level representation of primitive entities of the high level language. For example, the execution level definition at hand has the following properties.

1) A word size sufficiently wide to enable the representation of a LISP cell by one computer word where one half denotes CAR and the other denotes CDR (we assume that the left half represents CAR and the right half CDR).

2) The existence of general-purpose accumulators and an addressing structure that assigns accumulators low core addresses thereby enabling the representation of NIL by zero.

3) A hardware stack manipulation capability.

4) Two's complement arithmetic.

Every high level language has associated with it a set of control structures. Part of the execution level definition is a specification of conventions as to the implementation of these control structures. For example, in Algol [16] there is a great concern with respect to procedures and the display mechanism for establishing proper variable bindings. In LISP the primary control structure is the function call mechanism which acts as a conduit for information between program segments. This requires the establishment of conventions with respect to where such information is to be found. For example, in the execution level definition at hand, functions are called with arguments in accumulators 1 through N where N denotes the number of arguments. Results of functions are returned in accumulator 1. Each word containing an argument or a result has a format consisting of a zero in its left half and a LISP pointer in its right half. The stack is used to pass control information between functions—i.e., whenever a function call occurs, the return address is pushed on the stack. This implies the existence of a stack pointer which must be known to reside in a specified location (accumulator 12 in our case). Other LISP implementations often use the stack for passing arguments and control as well as returning results.

```
(DEFPROP NEXT (LAMBDA (L X)
        (COND ((OR (NULL L) (NULL (CDR L))) NIL)
              ((EQ (CAR L) X) (CAR (CDR L)))
              (T (NEXT (CDR L) X)))) EXPR)

NEXT(L,X) = if NULL(L) or NULL(CDR(L)) then NIL
            else if CAR(L) EQ X then CAR(CDR(L))
            else NEXT(CDR(L),X)
```

Fig. 2. LISP and MLISP encodings of NEXT.

```
NEXT    (JUMPE 1 TAG1)      jump to TAG1 if L is NIL
PC2     (MOVE 3 1)          load accumulator 3 with L
        (HRRZ 1 0 1)        load accumulator 1 with CDR(L)
        (JUMPE 1 TAG1)      jump to TAG1 if CDR(L) is NIL
        (HLRZ 4 0 3)        load accumulator 4 with CAR(L)
        (CAME 4 2)          skip if CAR(L) is EQ to X
        (JRST 0 PC2)        compute NEXT(CDR(L),X)
        (HLRZ 1 0 1)        load accumulator 1 with CAR(CDR(L))
TAG1    (POPJ 12)           return
```

Fig. 3. LAP encoding of NEXT.

IV. Example

In order to demonstrate the usefulness of compiler testing and the necessary machine description facilities, we give an example of the type of results that can be expected from such a system. However, we first present a brief definition of our subset of LISP.

Briefly, we are dealing with a subset of LISP which allows side effects and global variables. There are two restrictions. First, a function may only access the values of global variables or the values of its own local variables—it may not access another function's local variables. Second, the target label of a GO in a PROG must not have occurred physically prior to the occurrence of the GO to the label. The first restriction is motivated by the manner in which nonglobal variables are treated in compiled LISP—namely, they serve as placeholders for computations. The second restriction is due to the implementation at hand. It could be lifted if GO's were handled as function calls rather than branches. For more details see [19].

As an example, consider the function NEXT whose LISP 1.6 and MLISP [21] (a parentheses-free LISP also known as meta-LISP which is used throughout the paper) definitions are given in Fig. 2. The function takes as its arguments a list L and an element X. It searches L for an occurrence of X. If such an occurrence is found, and if it is not the last element of the list, then the next element in the list is returned as the result of the function. Otherwise, NIL is returned. For example, application of the function to the list (A B C D E) in search of D would result in E, while a search for E or F would result in NIL.

Fig. 3 contains a LAP encoding, obtained by a hand coding process, for the function given in Fig. 2. The format of a LAP instruction is (OPCODE AC ADDR INDEX) where INDEX and ADDR are optional. OPCODE is a PDP-10 instruction optionally suffixed by @ which denotes indirect addressing. The AC and INDEX fields contain numbers between 0 and decimal 15.

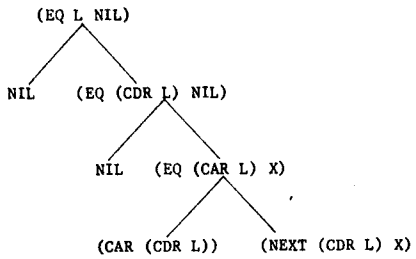

Fig. 4. Intermediate representation of Fig. 2.

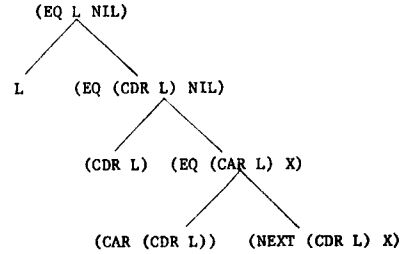

Fig. 5. Intermediate representation of Fig. 3.

ADDR denotes the address field. The meaning of the instructions used in the example LAP encoding should be clear from the adjoining comments.

Figs. 4 and 5 give the symbolic intermediate representations of the functions encoded by Figs. 2 and 3, respectively. Notice that the representation is in the form of a tree with a predicate at the root and the left and right subtrees correspond to the true and false values, respectively, of the predicate. Briefly, a proof of equivalence must show that the two representations can be transformed into each other. There are two discrepancies which involve the conclusion when the predicates (EQ L NIL) and (EQ (CDR L) NIL) are true. In both cases the results are valid since when these predicates are true, equality allows the interchanging of the arguments of such predicates in all subsequent use of these arguments. In addition, the proof procedure must demonstrate that (CDR L) need only be computed once, as in Fig. 3, rather than at three separate instances, as in Fig. 2. This is accomplished by use of an additional intermediate representation which reflects the instance at which each computation was performed.

V. Machine Description

The machine instructions are described via the use of procedures in a programming language quite similar in appearance to the class of register transfer languages [1] common in hardware descriptions. In the following discussion we present the description facility and show how it is used by means of examples. In the course of the presentation much use is made of LISP, the PDP-10, and the system reported in [19]. The discussion culminates with a demonstration of how the description facility coped with some unorthodox uses of instructions which were not included in that original system.

A. Description Facility

At the heart of the description process is the set of primitives which are provided for describing such basic operations as the operand fetch cycle, data transfer, control, predicate testing, and sense of tests. These primitives are encoded in a manner which binds the high level language in question to its execution level definition. The most important property of the primitives is that they are independent of the instructions that invoke them. Thus if another computer instruction set were to be described we would only need to make sure that architectural constraints were satisfied. If the latter were false, then some of the effected primitives would need to be recoded, e.g., a computer with a different effective address calculation method.

Our descriptive language is a subset of MLISP. We have chosen it largely because of the ease with which symbolic data can be manipulated and data structures defined. Also, MLISP being a subset of LISP enables the symbolic interpretation procedure to take advantage of the property of indistinguishability of program and data and use the EVAL mechanism of LISP to directly evaluate the low level language program. If we were dealing with low level languages of the type used in microprograms, then register transfer languages might be more appropriate (see [13] where a combination of VDL [17] and APL [9] are used).

The descriptive language is very much like Algol with the exception that the underlying data structure is the set of s-expressions. Variables are declared in a routine via the construct NEW and are local to the declaring procedure. There exists a global variable PCG which is used to keep track of the program counter. The symbolic interpretation system is aware of this variable and ensures that it is incremented as well as kept up to date when executing various branches of condition testing instructions. Predeclared constants are also available. These include X11 which is a word containing a data pointer of value 1 in each halfword, and ZEROCNST which is a word containing a data pointer of value 0 in each halfword.

An instruction is described by a one argument procedure of type FEXPR which has the same name as the instruction. The argument represents a list bound to the CDR of the LAP "word" containing the instruction. In other words the parameter to each procedure is a list comprising the accumulator, address, and index fields of the LAP word. Hence there is one procedure with indirect addressing and one without. The value of the accumulator field is accessed by the function ACFIELD, and the combined value of the address field modified by the contents of the accumulator denoted by the index field (if nonzero) is accessed by the function EFFECTADDRESS.

B. Examples

In order to demonstrate the instruction description process, we show how the instruction MOVE, HRRZ, PUSHJ, JUMPE, SKIPE, and TDZN are handled. The presentation also indicates how the symbolic interpretation system understands certain applications of the instructions given only the description seen here. Let AC denote the name of the accumulator specified by the accumulator field of the instruction.

MOVE is a simple instruction which is analogous to a load operation. It moves the contents of the effective address into accumulator AC. Fig. 6 shows its procedural description. Note

```
FEXPR MOVE(ARGS);
LOADSTORE(ACFIELD(ARGS),CONTENTS(EFFECTADDRESS(ARGS)));
```

Fig. 6. MOVE instruction.

```
FEXPR HRRZ(ARGS);
LOADSTORE(ACFIELD(ARGS),
        EXTENDZERO(RIGHTCONTENTS(EFFECTADDRESS(ARGS))));
```

Fig. 7. HRRZ instruction.

```
FEXPR PUSHJ(ARGS);
BEGIN
  NEW ADDRESS;
  ADDRESS←EFFECTADDRESS(ARGS);
  ALLOCATESTACKENTRY(ACFIELD(ARGS));
  ADDX(<ACFIELD(ARGS),X11>);
  LOADSTORERIGHT(RIGHTCONTENTS(ACFIELD(ARGS)),
                 FORMRETURNADDRESS(PCG));
  LOADSTORELEFT(RIGHTCONTENTS(ACFIELD(ARGS)),FLAGSPOINTER());
  UNCONDITIONALJUMP(ADDRESS);
END;
```

Fig. 8. PUSHJ instruction.

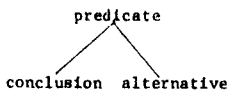

Fig. 9. Tree representation of a test.

```
FEXPR JUMPE(ARGS);
BEGIN
  NEW TST;
  TST←CHECKTEST(CONTENTS(ACFIELD(ARGS)),ZEROCNST);
  IF TST THEN RETURN(
    IF CDR TST THEN UNCONDITIONALJUMP(EFFECTADDRESS(ARGS))
    ELSE NEXTINSTRUCTION());
  TRUEPREDICATE();
  CONDITIONALJUMP(ARGS,FUNCTION JUMPETRUE);
  JUMPALTERNATIVE(ARGS,FUNCTION JUMPEFALSE);
END;

FEXPR JUMPETRUE(ARGS);
UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));

FEXPR JUMPEFALSE(ARGS);
NEXTINSTRUCTION();
```

Fig. 10. JUMPE instruction.

the use of the primitive LOADSTORE(A,B) whose effect is to store B in location A.

A slightly more complicated instruction is HRRZ. It is used to load the right half of accumulator AC with the right half of the contents of the effective address, and to clear the left half (see Fig. 7). When a LISP pointer resides in an accumulator, say B, then indexing via accumulator B results in the computation of CDR since if the effective address results in a LISP pointer, then its right half contains CDR. Note the use of the primitive EXTENDZERO which has a halfword as its argument and returns a word containing zero in its left half and the said argument in its right half. Detection of CAR and CDR operations is done by primitives such as CONTENTS, LEFTCONTENTS, and RIGHTCONTENTS, and a mechanism for keeping track of types associated with various locations (see Section VI for further details).

Some instructions may require a sequence of statements to describe their effect. For example, the PUSHJ instruction described in Fig. 8. Briefly, this instruction saves the incremented value of the program counter PCG on the stack which is pointed at by accumulator AC and continues execution at the location denoted by the effective address (i.e., a function call). Notice the procedural nature of the description. A more informal description is as follows. Allocate a stack entry, increment both halves of the stack pointer (the construct <. . .> denotes the LISP function LIST which has an arbitrary number of arguments), store the return address and a half word containing the processor flags on the stack, and continue execution at the location denoted by the effective address.

Notice the use of the primitive UNCONDITIONALJUMP to specify unconditional flow of control to the address specified by the argument. UNCONDITIONALSKIP and NEXTINSTRUCTION accomplish similar functions. These primitives are also responsible for detecting the occurrence of external function calls and recursion. Recall that in this case the function being called is not symbolically interpreted; instead, control resumes at the next instruction which is to be executed at the termination of the function call.

Condition testing instructions entail more complicated descriptions. In such cases we must first determine if the value of the condition is already known. This is accomplished by the primitive CHECKTEST which returns a value of NIL if the value of the condition is unknown, and the dotted pair (T.T) or (T.NIL) if the condition is known to be true or false, respectively. In the affirmative case, the appropriate path is taken and symbolic interpretation continues. If the value of the condition is unknown, then the sense of the test performed is recorded by the primitives TRUEPREDICATE and FALSEPREDICATE, the actual test is recorded, and the two alternate paths are symbolically interpreted in order and the result returned is a tree as shown in Fig. 9.

As examples of condition testing instructions we examine the instructions JUMPE, SKIPE, and TDZN. In all three cases we see either the pair of primitives CONDITIONALJUMP and JUMPALTERNATIVE, or CONDITIONALSKIP and SKIPALTERNATIVE. They are used to invoke the symbolic interpretation process for the true and false cases of a condition whose value is unknown. The arguments indicate the original parameter to the procedure corresponding to the instruction and the name of the routine used to execute the remainder of the condition. The routine is necessary for the purpose of indicating flow of control as well as any further properties of the instruction that depend on the result of the test. The JUMPE instruction (see Fig. 10) is used to conditionally jump to the effective address if the contents of accumulator AC is zero.

In an execution level definition where NIL is represented by zero, the JUMPE instruction provides a fast way of testing against NIL. For example, after symbolic interpretation of the (JUMPE 1 TAG1) instruction at label NEXT in Fig. 3, and before activation of CONDITIONALJUMP and JUMPALTERNATIVE, we have the partial tree given in Fig. 11,

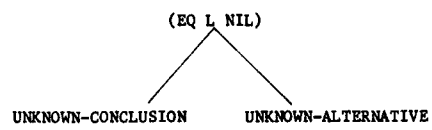

Fig. 11. Result of symbolic interpretation of first (JUMPE I TAG1) in Fig. 3.

```
FEXPR SKIPE(ARGS);
BEGIN
  NEW MEMG,TST;
  MEMG←CONTENTS(EFFECTADDRESS(ARGS));
  IF ACFIELD(ARGS) NEQ 0 THEN LOADSTORE(ACFIELD(ARGS),MEMG);
  TST←CHECKTEST(MEMG,ZEROCNST);
  IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALSKIP()
                     ELSE NEXTINSTRUCTION());
  TRUEPREDICATE();
  CONDITIONALSKIP(ARGS,FUNCTION SKIPETRUE);
  SKIPALTERNATIVE(ARGS,FUNCTION SKIPEFALSE);
END;

FEXPR SKIPETRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR SKIPEFALSE(ARGS);
NEXTINSTRUCTION();
```

Fig. 12. SKIPE instruction.

```
FEXPR TDZN(ARGS);
BEGIN
  NEW ACG,MEMG,TST;
  MEMG←CONTENTS(EFFECTADDRESS(ARGS));
  ACG←CONTENTS(ACFIELD(ARGS));
  LOADSTORE(ACFIELD(ARGS),SETMASKEDBITS(ACG,MEMG,0));
  TST←CHECKTEST(ACG,MAKEMASK(MEMG));
  IF TST THEN RETURN(IF CDR TST THEN NEXTINSTRUCTION()
                     ELSE UNCONDITIONALSKIP());
  FALSEPREDICATE();
  CONDITIONALSKIP(ARGS,TDZNTRUE);
  SKIPALTERNATIVE(ARGS,TDZNFALSE);
END;

FEXPR TDZNTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR TDZNFALSE(ARGS);
NEXTINSTRUCTION();
```

Fig. 13. TDZN instruction.

i.e., (EQ L NIL). Note the use of UNKNOWN-CONCLUSION and UNKNOWN-ALTERNATIVE to indicate that the true and false cases of the condition have not yet been symbolically interpreted.

The SKIPE instruction (see Fig. 12) skips the next instruction if the contents of the effective address is zero. If the accumulator name specified by the accumulator field is nonzero, then the said accumulator is loaded with the contents of the effective address. Note the use of a conditional statement in the description of the instruction to determine if the accumulator field is zero.

The TDZN instruction (see Fig. 13) results in zeroing the bits in accumulator AC corresponding to bits that are 1 in the mask word contained in the effective address. In addition, the next instruction is skipped if any of the bits selected by the mask word in AC were 1.

TDZN is an interesting instruction since it makes use of bit masks. The primitive SETMASKEDBITS interprets its second argument as a mask and sets the corresponding bits in its first argument to the bit value denoted by its third argument. MAKEMASK is a primitive which converts its argument to a bit mask. The CHECKTEST primitive is sufficiently clever (by use of data types, as shown in Section VI) to perform the appropriate comparison operation. Namely, in the case of a bit test, the mask specifies which bits are to be zero and thus the test is one of selective equality. Thus in some sense, the primitive MAKEMASK serves to indicate to the CHECKTEST primitive that the predicate will involve a bit comparison.

The TDZN instruction can be used to implement a NOT function with two instructions in an execution level definition where NIL is represented by zero. When used with a mask containing a 1 in every bit position (i.e., a value of -1), TDZN will set the tested accumulator to zero as well as skip the next instruction if the former contents of the tested accumulator were nonzero. This is equivalent to setting a location to NIL and branching if it was previously non-NIL. If the former contents were indeed NIL, then the next instruction can load the accumulator with the atom corresponding to true (i.e., T).

```
FEXPR JRA(ARGS);
BEGIN
  LOADSTORE(ACFIELD(ARGS),CONTENTS(LEFTCONTENTS(ACFIELD(ARGS))));
  UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));
END;
```

Fig. 14. JRA instruction.

C. Extension

When the compiler testing system in [19] was originally designed, only the machine instructions which were thought to be useful in handling LISP were described. Subsequently, as a means of testing the adequacy of the description facility, questions were posed of the nature "suppose a certain instruction was used in a particular manner to achieve a desired effect." In this section we demonstrate how the description facility coped with some unorthodox uses of two instructions, JRA and TLNN, which were not in the original system. Two items are worthy of note here. First, nowhere in the descriptions of the instructions do we make any provisions for their use in the said manner to achieve the desired effect. Second, no new primitives are needed to handle these examples although we suspect that in other cases there may be a need for some additional primitives or modifications to existing ones.

The JRA instruction (see Fig. 14) results in placing the contents of the location addressed by the left half of accumulator AC into the same accumulator. Execution continues at the effective address.

Now, suppose the instruction (JRA AC LABEL) has been seen, where LABEL is the immediately following instruction and accumulator AC contains the contents of a LISP cell, say A. In other words, accumulator AC contains pointers to CAR(A) and CDR(A) in its left and right halves, respectively. The effect of this particular instance of the instruction is to load the left and right halves of accumulator AC with CAR(CAR(A)) and CDR(CAR(A)), respectively, and continue execution at the following instruction. This effect is detected automatically as a result of the symbolic interpretation of JRA, since LEFTCONTENTS of A is CAR(A) and CONTENTS of a LISP cell pointed at by CAR(A) is a word containing CAR(CAR(A)) and CDR(CAR(A)) in its left and right halves, respectively. LOADSTORE will ensure that accumulator AC now contains

```
FEXPR TLNN(ARGS);
BEGIN
  NEW TST;
  TST←CHECKTEST(LEFTCONTENTS(ACFIELD(ARGS));
                MAKEMASKHALF(EFFECTADDRESS(ARGS)));
  IF TST THEN RETURN(IF CDR TST THEN NEXTINSTRUCTION()
                     ELSE UNCONDITIONALSKIP());
  FALSEPREDICATE();
  CONDITIONALSKIP(ARGS,TLNNTRUE);
  SKIPALTERNATIVE(ARGS,TLNNFALSE);
END;

FEXPR TLNNTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR TLNNFALSE(ARGS);
NEXTINSTRUCTION();
```

Fig. 15. TLNN instruction.

these values. The immediately following instruction is the next instruction to be executed by virtue of the primitive UNCONDITIONALJUMP.

The TLNN instruction (see Fig. 15) results in a skip of the next instruction in sequence if the bits in the left half of accumulator AC corresponding to the bits that are 1 in the mask formed by the effective address are not all equal to zero. Otherwise, the next instruction in sequence is processed.

Now, suppose the instruction (TLNN AC -1) has been seen where accumulator AC contains the contents of a LISP cell, say A. In other words AC contains pointers to CAR(A) and CDR(A) in its left and right halves, respectively. The effect of this particular instruction is to test if CAR(A) is EQ to NIL and skip the next instruction if false. This effect is detected automatically as a result of the symbolic interpretation of TLNN, since LEFTCONTENTS of a LISP cell, say A, is CAR(A) and a test of equality against a bit mask containing all 1's is identical to checking if all of the bits in CAR(A) are zero, or equivalently if CAR(A) is NIL in an execution level definition where NIL is represented by zero. It should be noted that we have already seen a similar line of reasoning applied with respect to an application of the TDZN instruction.

VI. Memory

In the previous section we saw a description facility for machine instructions. We also hinted at an understanding process. This was evidenced by such examples as TDZN, TLNN, and JRA which showed how instructions were used to accomplish results not at all obvious from the instruction descriptions. The real work in building the intermediate representation is done by primitives such as EFFECTADDRESS, CONTENTS, UNCONDITIONALJUMP, etc. These primitives are used to encode the effects of the instruction on a computation model. This computation model reflects the contents of memory at all times and is directly dependent on the execution level definition of the high level language. In this section we describe what we loosely term the memory. This description consists of the actual data structures constituting the memory, the various data types that can be stored in the memory, and the mechanism by which we keep track of the contents of the memory.

Assuming a separation of data and program, data memory can be further partitioned into internal and environmental data structures. The former correspond to machine related constructs such as the accumulators and the stack, while the latter reflect the constructs which owe their existence to the high level language. In the case of LISP, the environmental data structures include the List Structure and the cells containing the values of the global variables.

An alternative view of memory is one consisting of two parts, locations that can be overwritten and those that cannot. The latter part consists of the area containing the program to be executed. Clearly, overwriting should be forbidden since in our proofs we assume that a program remains the same. Otherwise an equivalence proof is somewhat meaningless due to the possible recursive nature of the functions with which we are dealing. Thus we do not allow self-modifying code. Note that the contents of all locations in memory may be read. The locations that can be overwritten include the accumulators, the stack, and the environmental data structures. The restriction on overwriting some of the environmental data structures is that they may only be overwritten with data of the high level language. In the case of LISP this means that elements in the List Structure may only be overwritten with LISP pointers.

The stack and some of the accumulators are useful as temporary data areas. This dictates a need for a mechanism for keeping track of the contents of various locations. This is of utmost importance for the primitives which are used to describe the various instructions of the target computer since their actions depend to a large degree on the data types associated with their operands. Specifically, when compositions of operations are performed, we must have a meaningful way of expressing the intermediate results. This is accomplished by associating with each half word a data type as well as a value. These types enable us to describe the contents of the various locations in a manner that renders subsequent operations meaningful when using them as data.

Environmental and internal data types are related in a manner analogous to environmental and internal data structures. The environmental data types typically refer to values which have a corresponding interpretation in the high level language in question. In a language such as Pascal [24] these would include structure pointers and array pointers, among others. In the case of LISP (which is typeless) we only have one environmental data type, namely the LISP pointer. Initially, the only locations containing data of type LISP pointer are the accumulators containing the parameters to the function being symbolically interpreted and the cells containing the global variables. As the LAP program is symbolically interpreted all LISP functions of LISP pointers result in LISP pointers. In addition, each cell pointed at by a LISP pointer, say A, contains CAR(A) in the left half and CDR(A) in the right half. Thus whenever the left half of a cell pointed at by a LISP pointer is accessed, CAR of the LISP pointer is computed, and similarly for the right half and the CDR operation.

Internal data types refer to constructs which owe their existence to the particular target machine and not to the high level language. In the case of the PDP-10, they include such items as stack pointers, bits, labels, address constants, half words (the index, accumulator, opcode, and indirect addressing fields in an instruction), numbers, and unknown (locations about whose contents nothing is known). The concept of internal

data types is important because it provides a capability to detect illegal operations by virtue of a mismatch of types of operands. In the following paragraphs we describe briefly some of these types as they relate to LAP as well as some properties common to groups of types.

A stack pointer is a data structure having two fields—one denoting a count and the other an address. Therefore, the stack pointer data type has two subtypes. One for the count which originally appears in the left half and one for the address which appears in the right half. When symbolic interpretation of a function begins, the values of both the stack count and stack address are (relative) zero. This enables the detection of illegal accesses to locations belonging to the caller (i.e., negative stack address), locations not yet allocated (i.e., keep track of maximum number of stack entries allocated), and ensures that the stack depth is the same at function exit as it was at function entry. Furthermore, computations involving stack addresses can be performed. This is particularly useful when a stack is deallocated via use of arithmetic operations rather than the customary POP operation, since resulting illegal stack pointers (i.e., negative values) can be detected.

Non-LISP numbers and symbolic addresses of instructions are represented by the same data type. Numbers and symbolic addresses can be combined via addition and subtraction to form new addresses, and symbolic addresses can be subtracted from each other in which case a number results. There is special treatment of the numeric constant -1 which corresponds to all bits being 1 in a word and is useful in handling borrow terms in subtraction. No confusion can arise between LISP numbers and regular numbers because the former appear QUOTEd in a LISP program as they correspond to atoms which are not to be evaluated.

The bit type is useful for describing the contents of words for bit testing operations. Bit tests using a mask, where the mask stipulates which bits are to be zero, are basically no different than regular tests (i.e., a test for equality against zero). The only difference is that the effective address or its contents indicates which bits are to be zero. With this in mind, an operation making use of a mask containing 1's in the positions which are to be tested, is represented as a sequence of NIL and 0. The latter denotes that the bit is to be zero while the former denotes a "don't care" condition. Now, it is clear that a bit testing operation is analogous to a test against 0 with the selected bits tested rather than an entire word or half word. Note that the two's complement representation of numbers implies added significance for -1, since -1 corresponds to a mask with all bits being 1 and if used in a masked test it results in a test against zero (NIL in the execution level definition at hand).

The various internal data types also have some properties in common. In the case of stack pointers, labels, and unknown, the data type information is particularly useful since it provides a mechanism for keeping track of arithmetic operations. Basically, in these cases the value associated with the item in question acts as a relocation constant and the normal conventions governing the behavior of assemblers with respect to relocation arithmetic also hold. In the case of bits, half words, and instructions the value associated with a specific data type merely acts as a different representation of a number with an allowance for conversion between the synonyms. Address constants are somewhat dual to unknown since they represent a location whose address is unknown while at the same time its contents are known.

VII. Conclusion

The procedural nature of our description facility for a computer instruction set and the use of primitives are the key distinguishing factors between our method for compiler testing and those employing decompilation methods. This is evidenced by the absence in the instruction descriptions of any information as to how particular constructs in the high level language are encoded. We saw that the instruction description simply indicates the effect that the particular instruction has on the computation model.

At present, a system [19] exists that proves the correctness of translations of LISP 1.6 programs on a PDP-10. Future work would include the extension of the ideas presented here to a LISP implementation on an architecturally different computer. For example, a PDP-11 [4] which has a considerably smaller word size thereby forcing a LISP implementation where two words are necessary to contain a LISP cell. This would mean that a LISP cell would most likely be represented by a block of two contiguous words. Other interesting properties of a PDP-11 implementation include a different addressing structure, condition codes, and separate function control and parameter stacks to enable a more efficient use of the limited address space. This would make use of memory management by keeping programs and data in separate address spaces. Also NIL would not be represented by zero since low core addresses do not serve as accumulators.

An equally interesting extension, although conceivably more difficult, is the adaptation of the ideas presented here to a different high level language. For example, in an algebraic language such as Pascal, we would have to model other data types such as arrays and structures. In such a case we would need to formulate an execution level definition which would indicate the representation taken by these constructs (e.g., array headers, etc.).

Regardless of the direction that such extensions take, they will undoubtedly provide insights into any deficiencies in the primitives used for machine description in addition to aiding in the identification of missing ones.

Acknowledgment

Special thanks go to Prof. V. G. Cerf for his constant advice and encouragement during a period in which some of this research was pursued.

References

[1] G. Bell and A. Newell, *Computer Structures: Readings and Examples.* New York: McGraw-Hill, 1971.

[2] L. P. Deutsch, "An interactive program verifier," Ph.D. dissertation, Dep. Comput. Sci., Univ. California, Berkeley, CA, May 1973.

[3] *PDP-10 System Reference Manual*, Digital Equipment Corp., Maynard, MA, Dec. 1969.

[4] *PDP-11 Reference Manual*, Digital Equipment Corp., Maynard, MA, Dec. 1973.

[5] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Applied Mathematics*, J. T. Schwartz, Ed., vol. 19, American Mathematical Society, 1967, pp. 19-32.
[6] C. R. Hollander, "Decompilation of object programs," Ph.D. dissertation, Dep. Elec. Eng., Stanford University, Stanford, CA, Digital Syst. Lab. Tech. Rep. 54, 1973.
[7] J. C. Huang, "An approach to program testing," *Comput. Surveys*, vol. 7, pp. 113-128, Sept. 1975.
[8] S. S. Husson, *Microprogramming: Principles and Practices*. Englewood Cliffs, NJ: Prentice-Hall, 1970.
[9] K. E. Iverson, *A Programming Language*. New York: Wiley, 1962.
[10] J. King, "A program verifier," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1969.
[11] J. A. N. Lee, *Computer Semantics*. New York: Van Nostrand Reinhold, 1972, pp. 346-347.
[12] G. B. Leeman Jr., W. C. Carter, and A. Birman, "Some techniques for microprogram validation," in *Proc. 1974 IFIP Cong.*, pp. 76-80.
[13] G. B. Leeman Jr., "Some problems in certifying microprograms," *IEEE Trans. Comput.*, vol. C-24, pp. 545-553, May 1975.
[14] R. L. London, "The current state of proving programs correct," in *Proc. Ass. Comput. Mach. 25th Annu. Conf.*, 1972, pp. 39-46.
[15] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *Commun. Ass. Comput. Mach.*, vol. 3, pp. 184-195, Apr. 1960.
[16] P. Naur, Ed., "Revised report on the algorithmic language ALGOL 60," *Commun. Ass. Comput. Mach.*, vol. 3, pp. 299-314, May 1960.
[17] F. J. Neuhold, "The formal description of programming languages," *IBM Syst. J.*, vol. 10, pp. 86-113, 1971.
[18] L. H. Quam and W. Diffie, "Stanford LISP 1.6 manual," Dep. Comput. Sci., Stanford Univ., Stanford, CA, Stanford Artificial Intelligence Project Operating Note 28.7, 1972.
[19] H. Samet, "Automatically proving the correctness of translations involving optimized code," Ph.D. dissertation, Dep. Comput. Sci., Stanford Univ., Stanford, CA, Stanford Artificial Intelligence Project Memo AIM-259, 1975.
[20] —, "A normal form for LISP programs," Dep. Comput. Sci., Univ. Maryland, College Park, MD, TR-443, 1976.
[21] D. C. Smith, "MLISP," Dep. Comput. Sci., Stanford Univ., Stanford, CA, Stanford Artificial Intelligence Project Memo AIM-135, Oct. 1970.
[22] N. Suzuki, "Verifying programs by algebraic and logical reductions," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 473-481.
[23] T. Winograd, "Procedures as a representation for data in a computer program for understanding natural language," Massachusetts Inst. Tech., Cambridge, MA, MAC TR-84, Feb. 1971.
[24] N. Wirth, "The programming language PASCAL," *Acta Informatica*, vol. 1, pp. 35-63, 1971.

Hanan Samet (S'70-M'75) received the B.S. degree in engineering from the University of California, Los Angeles, and the M.S. degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

Since 1975 he has been an Assistant Professor of Computer Science at the University of Maryland, College Park. His research interests are data structures, programming languages, code optimization, and data base management systems.

Dr. Samet is a member of the Association for Computing Machinery, SIGPLAN, Phi Beta Kappa, and Tau Beta Pi.

Deletions That Preserve Randomness

DONALD E. KNUTH

Abstract—**This paper discusses dynamic properties of data structures under insertions and deletions. It is shown that, in certain circumstances, the result of n random insertions and m random deletions will be equivalent to $n-m$ random insertions, under various interpretations of the world "random" and under various constraints on the order of insertions and deletions.**

Index Terms—**Analysis of algorithms, binary search trees, data organization, deletions, priority queues.**

Manuscript received December 10, 1976; revised March 18, 1977. This research was supported in part by the National Science Foundation under Grant MCS 72-03752 A03, by the Office of Naval Research under Contract N00014-76-C-0330, and by the IBM Corporation. Reproduction in whole or in part is permitted for any purpose of the United States Government.

The author is with the Department of Computer Science, Stanford University, Stanford, CA 94305.

I. Introduction

When we try to analyze the average behavior of algorithms that operate on dynamically varying data structures, it has proved to be much easier to deal with structures that merely grow in size than to deal with structures that can both grow and shrink. In other words, the study of insertions into data structures has proved to be much simpler than the study of insertions mixed with deletions. One instance of this phenomenon is described in [5], where what looks like an especially simple problem turns out to require manipulations with Bessel functions, although the data structure being considered never contains more than three elements at a time.

Occasionally an analysis of mixed insertions and deletions turns out to be workable because it is possible to prove some sort of invariance property; if we can show that deletions pre-