CHAPTER 11

REUSABILITY OF DESIGN FOR LARGE SOFTWARE SYSTEMS: An Experiment with the SETL Optimizer

ED DUBINSKY
Department of Mathematics, Clarkson College

STEFAN FREUDENBERGER
Multiflow Computer, Inc.

EDITH SCHONBERG *and* **J. T. SCHWARTZ**
Computer Science Department, New York University

11.1 A BRIEF OVERVIEW OF THE EXPERIMENT

Reusability of design is the ability to convey the overall procedural structure and principal data designs of large complex programs to those who wish to develop the same or similar functions in new environments. This is an important aspect of program reusability overall. Design reuse will frequently be associated with increased efficiency requirements; for example, a programming system designed for a computer of a certain size may have to be redeveloped for a significantly smaller machine, or a program used only occasionally

or only during development and testing may be accepted for production use and have to be made more efficient. For these purposes, an appropriate software design tool is a specification language for fully describing all significant abstract features of complex programs, but suppressing as many design-irrelevant details as possible.

This paper reports on a significant experiment in the expression and reuse of a program design. The vehicle used for transmission of design is an executable, procedural, but very high-level language called SETL, developed at New York University [Schwartz *et al.*, 1986; Dewar *et al.*, 1979]. The goal was to significantly improve the efficiency (at least by an order of magnitude) of a large and complex program written in SETL simply by translating it into a lower-level language. The translation methodology aimed for may be described as manual but mechanical; neither the algorithms nor overall program structure were to be changed. The program translated in the experiment is the SETL optimizer, a part of the SETL system itself.

This experiment in two-step program development shows that the high-level specification language provides a good tool not only for experimenting with algorithms, but also for clearly documenting design; the translated version is markedly more efficient, and the code is well organized and clean; the translation process proceeds quickly and accurately, with fewer bugs than if developed from scratch. In short, the postponement of low-level implementation decisions until the end of a large software development project enhances the final product.

SETL, the source language in our experiment, has general finite sets, maps/relations, and tuples as its basic data types. In addition to conventional control structures, it implements more sophisticated operations, such as universal and existential quantification over sets. The language is weakly typed, but optionally a data representation sublanguage (DRSL) can be used to provide information on the desired representation of variables. The DRSL serves both to improve the efficiency of SETL programs and to document the use of variables. We have found that the DRSL plays an important role in the manual translation of SETL; more precisely, it suggests data structure implementations for SETL objects.

The target language was LITTLE, a lower-level language whose semantics lies somewhere between FORTRAN and C. Its basic data type is the bit string, which may be accessed directly and also interpreted as an integer, Boolean, real, or character string. The only available data structure in LITTLE is the one-dimensional static array. Control structures are conventional and comparable to those of Pascal, except that LITTLE does not support recursion. Although LITTLE has no record structures, it has a macro definition facility that is heavily used to improve code readability and enables programmers to emulate records.

The SETL optimizer/analyzer is a sophisticated package that performs global (interprocedural) analysis of SETL programs and, on the basis of this analysis, attempts ambitious code improvements [Freudenberger, 1984;

Freudenberger *et al.*, 1983]. Novel features of the optimizer include complex algorithms for inter- and intraprocedural data flow analysis [Schwartz and Sharir, 1979], automatic variable type detection [Tenenbaum, 1974], and automatic selection of run-time data structures for sets and maps [Schonberg, Schwartz, and Sharir, 1979]. It uses the full expressive power of SETL, including the use of deeply nested sets, maps, and tuples, and complex set-former expressions. The optimizer, developed over a period of four years, consists of about 15,000 lines of SETL code.

Even though the rest of the SETL system is in LITTLE, SETL was chosen as the initial implementation language for the optimizer for a number of reasons. The optimizer served as a research testbed for experimenting with new algorithms; SETL facilitates prototyping and program modification, which are important for algorithm testing. The DRSL enables experimentation with different set representations without modifying any code. The ability to write algorithms in a lucid and concise manner was also valuable, both for joint project work and for inclusion of code in reports.

However, because of its size and slowness, the SETL version can only be used to optimize small programs (about 300 lines in 25 minutes). On the other hand, the optimized results from these small test programs indicated that a production-quality implementation was worth building. Thus the rewrite project had a very practical goal: to improve the optimizer's speed so that it could be applied to large SETL programs, and actually serve as a useful tool for SETL programmers. For another example of the implementation of a large software system from a SETL specification, see [Schonberg and Schields, 1986].

Section 11.2 gives a brief summary of SETL. Section 11.3 presents issues arising in the translation project. Sections 11.4, 11.5, and 11.6 give, in order of increasing complexity, three examples of data structure choices made in LITTLE that were guided by the DRSL declarations in SETL; these examples illustrate how the high-level design is reused. The conclusion, Section 11.7, gives specific performance results and summarizes our viewpoint.

11.2 SETL BASICS

Detailed descriptions of SETL can be found in [Dewar *et al.*, 1979] and [Schwartz *et al.*, 1986]. We limit ourselves here to the most salient features of the language and those features that are most significant in the SETL version of the optimizer.

SETL is an imperative, sequential language with assignment, weak typing, dynamic storage allocation, and the usual atomic types: numeric types, Booleans, strings, and generated atoms. (An atom is simply a token that is distinct from all other atoms; the only operation among atoms is equality comparison.) Declarations are optional: The data representation sublanguage (Section 11.2.2) may be used to facilitate the choice of suitable data structures

for SETL objects. The composite types of SETL—*sets, tuples*, and *maps*—give the language its expressive power. We proceed to describe these.

11.2.1 The Types of SETL

Sets. Sets in SETL have the standard mathematical properties of sets: They are unordered collections of values of arbitrary types, containing no duplicate values, and the usual operations are defined on these collections: membership, union, intersection, range and domain, power set construction, and so on. Set iterators and constructors are built-in control structures that operate on sets. For example,

```
{x in S | P(x)}
```

denotes a subset of the set S, all of whose members x satisfy the predicate $P(x)$. Quantified expressions over sets use the quantifiers of first-order predicate calculus and describe common search constructs. For example,

```
exists x in S | P(x)
```

is a Boolean-valued expression whose value is **true** if some member of S satisfies the predicate P. As a side effect of its execution, the existentially quantified expression assigns to the bound variable x the value of some member of S that satisfies $P(x)$, if one exists.

A compound statement of the form

```
loop forall x in S | P(x) do
                        statements
                    end forall;
```

iterates over all elements x of S that satisfy $P(x)$, and executes *statements* for each iteration.

Tuples. Tuples in SETL are ordered sequences of arbitrary size, indexed by positive integers. As with all SETL composite types, the components of a tuple can be of arbitrary types, so that sets of tuples, tuples of sets, tuples of tuples of sets of integers, and so on, can be constructed. Insertions and deletions from either end allow tuples to be used as stacks and queues; concatenation makes tuples akin to lists for certain purposes. Tuple constructors and iterators are similar to the corresponding constructs for sets. For example:

```
[x: x in [2..100] | not exists y
                        in [2..x-1] | x mod y = 0]
```

constructs (inefficiently) the sequence of primes smaller than 100.

Maps. Maps in SETL faithfully reproduce the notion of mapping in set theory: A map is a set of ordered pairs, that is, tuples of length 2; the first components of these tuples constitute the *domain*. Both the domain and range of a map can be of arbitrary types. Maps can be used as tabular functions, and the *application* $M(x)$ can be evaluated (a map is an associative structure) or be the target of an assignment. Finally, mappings can be multivalued (relations) or single-valued (functions) and the image set $M\{x\}$ of a value x under the mapping M can also be retrieved and assigned to.

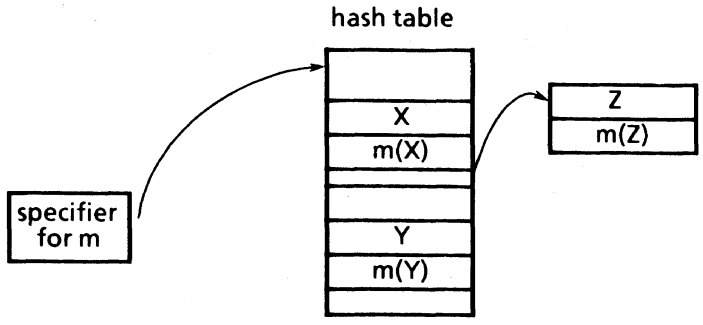
These familiar notions constitute the core of SETL. What distinguishes SETL from purely mathematical discourse is the requirements of executability. To guarantee that all valid SETL constructs are executable, only finite objects can be denoted.

11.2.2 Data Representation Sublanguage

At runtime, every object in SETL has a fixed length specifier that stores its type and, if the object is simple, its value, or, if otherwise, a pointer to its value. The elements of a set S are stored in a hash table referenced by the specifier of S. The elements of a tuple T are stored in an array referenced by the specifier of T. The elements of both sets and tuples are themselves specifiers, and may in turn point to other blocks. Since sets of two-tuples are maps, two-tuples are hashed on their first component to facilitate map retrieval. Operations such as set insertion, deletion, and membership test and map retrieval cost a hash operation. Figure 11.1 shows the default representation for a single-valued map M.

Through optional DRSL declarations, it is possible to obtain more efficient representations for sets and maps. A distinctive feature of the DRSL is that relationships among different variables may be described. The result

FIGURE 11.1
DEFAULT REPRESENTATION FOR MAPS. X AND Z HAVE SAME HASH-CODE



is, for example, that objects may internally store pointers to other objects, subsets may be represented as bit strings, and a form of record structure is supported, thus saving on hashing operations. Next we summarize the more important structures of the sublanguage.

The fundamental domains of a program, for example, sets that are typically the domains and ranges of maps, are declared as *base sets*. A program variable X may then be *based on* a base set. If X is a simple object, then the values assigned to X range over the elements of the base; if X is a set, a map, or a tuple, then the values of the components of X range over the elements of the base. A base set itself is not a program variable, and does not appear explicitly in the executable code; elements are added to the base according to the values that based variables take on. The representation of the based objects depends on how they are declared, as illustrated in the following examples.

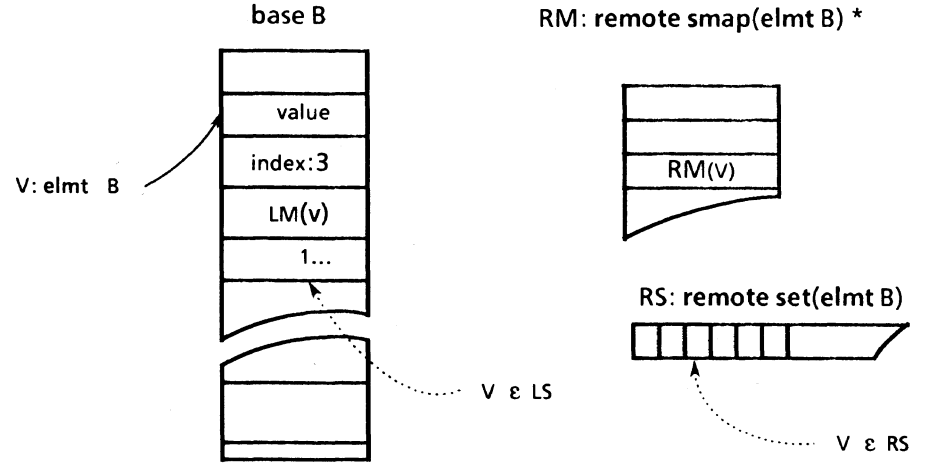
In the **repr** statement below, B is declared to be a base set of integers, and variable V to be an *element* of the base B.

```
repr     B:      base (integer);
         V:      elmt B;
end;
```

When V is assigned a value, the value is added to the base if it is not there already, and V stores a pointer directly to the value's element block.

Sets and maps based on B may be either *local* or *remote*. For locally based objects, component data is stored directly within the element blocks of the base (see Fig. 11.2). More specifically, for a local single-valued map LM with

FIGURE 11.2
BASE *B* AND OBJECTS BASED ON *B*



domain B, the range value LM(V) is stored in the element block of V. For a local set LS declared as a subset of B, each element block of B has a one-bit flag indicating whether the element is in LS. Local variables LM and LS are declared as follows:

```
repr    LM:     local smap (elmt B) integer;
        LS:     local set (elmt B);
end;
```

Local basings therefore provide a means of implementing a record structure for domains and associated attributes.

Remote objects, on the other hand, are stored in separate sequential data structures and not in the base itself. Each element in the base does, however, store a distinct index value, which is used as an object offset for the element in the remote structure. For example, consider a remote map RM, declared

```
RM:     remote smap (elmt B) integer;
```

The range of RM is stored as an array of integers (see Fig. 11.2). To retrieve RM(V), the index stored in the element block for V is used as an array index. Remote maps may have shorter lifetimes than the base.

A remote set is stored as a bit string. Set union and intersection operations are particularly efficient on remote sets. The variable RS declared below is a subset of B:

```
RS:     remote set (elmt B);
```

Each bit position in the bit string RS corresponds to an element in B and indicates membership of that element in RS. The index value of the element specifies the bit position of the element.

A **mode** declaration is provided in DRSL for defining frequently used types. For example, the following statements are equivalent to the declarations above:

```
repr
                B:      base (integer);
        mode    el_B:    elmt B;

                V:      el_B;
                LM:     local smap (el_B) integer;
                RM:     remote smap (el_B) integer;
                LS:     local set (el_B);
                RS:     remote set (el_B);
        end;
```

Such a set of **repr** declarations profile the types, relationships, and uses of program variables.

11.3 THE TRANSLATION PROCESS: PROBLEMS AND APPROACHES

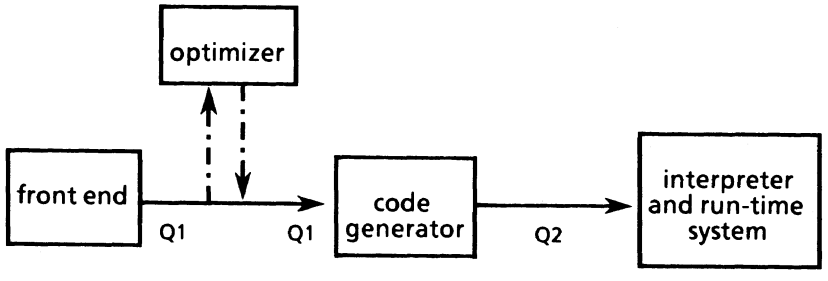
The SETL system (Fig. 11.3) includes a front end, which translates SETL source programs into an intermediate quadruple form (called Q1); a code generator, which translates Q1 into interpretable quadruples (called Q2), using DRSL or optimizer-specified information to choose instructions; an interpreter, which executes the Q2 instructions; and a run-time library, which implements SETL objects and operations and manages run-time storage. The optimizer is an optional pass before code generation and consists of 14 separate modules. It inputs and outputs Q1 quadruples. Since analysis is interprocedural, the optimizer must accommodate an entire source program at once.

In addition to various classical optimizations, such as code motion, redundant subexpression elimination, removal of unreachable code, and constant propagation, the optimizer also

1. determines the types of variables, and selects data structure representations for sets and maps, according to the way variables are used;
2. determines which routines are recursive and which temporaries and local variables can be made static; and
3. determines when copies must be made or may be suppressed as part of an instruction. (SETL has value semantics; i.e., objects are logically copied on assignment.)

Some of the initial problems encountered in translating the optimizer are common to all large program translation efforts. The group involved in the translation were not involved in writing the original SETL optimizer. In fact, the rewrite team was split geographically (between New York University and

FIGURE 11.3
STRUCTURE OF THE SETL SYSTEM



Clarkson College), making communication much harder. Additionally, the optimizer is a highly complex program—not because of poor coding style or lack of documentation, as is often the case, but because it includes intricate and novel algorithms. Finally, the target language LITTLE is awkward at best (its choice being dictated more for historic than good technical reasons). The lack of typing, records, recursion, and separate compilation protection, which are available in other systems languages such as C, Pascal, and Ada, exacerbated the difficulty of the task.

On the other hand, as a high-level specification, the SETL version is lucid and readable. Moreover, since it is an executable specification, it has been subjected to substantial testing and is trustworthy. Debugging the separate passes of the LITTLE version was greatly facilitated because the output could be compared with the output from the SETL version. Finally, the interfaces between modules are clean and well documented, so each module could be redeveloped independently.

The two main translation issues were (a) choosing data structures in LITTLE to implement abstract SETL objects (sets, tuples, maps)—with a suitable data structure interface, actual code transcription is straightforward—and (b) what to do about storage allocation—SETL is fully dynamic and runs in a garbage-collected environment, while LITTLE, like FORTRAN, is completely static.

A significant concern in the design was efficient use of space, in particular because the data for entire source programs is stored all at once and a great deal of information is generated during analysis. Therefore it is important to have both compact representations of objects and the ability to reclaim space no longer in use.

An appealing and seemingly convenient option, which was tried initially and later abandoned, is to reuse the storage management and set manipulation routines already written in LITTLE—namely, the routines in the SETL run-time library itself. The storage manager includes a compacting garbage collector; library routines are available for creating and manipulating SETL objects in various representations (those defined in the DRSL). The advantage of this approach is the possibility of a one-to-one correspondence between sets and tuples in the SETL version and in the LITTLE version and of using the same LITTLE code to implement these corresponding structures; translation would become completely mechanical, and debugging by comparison would be facilitated.

This approach proved too awkward and inefficient for the production quality results that we wanted, for the following reasons.

1. The SETL run-time library itself is very large (100,000 lines).
2. It is hard to use the run-time library interface directly, since it is not designed for an end user, but to be called by an interpreter for operations on single quadruples.

3. It is equally hard to interface directly with a garbage collector. Since the invocation of a garbage collection is unpredictable, pointers can be easily corrupted, unless the whole environment is designed to be protected.
4. Because of their generality, SETL data structures are bulky and wasteful of storage.

Instead, we chose to implement a very simple stack-based allocation scheme, together with a heap manipulated by explicit *allocate* and *free* functions. The SETL object translations were guided very strongly by the DRSL declarations in the SETL version. Section 11.4 describes a straightforward conversion of SETL objects into tables—the vast majority of optimizer structures followed this pattern. Sections 11.5 and 11.6 describe more complex examples of representation choices.

11.4 TABULAR REPRESENTATION OF DATA

The most common paradigm for data in the SETL version of the optimizer is a *domain* of objects with various *attributes* defined on the domain. Domains are represented as sets of atoms. Attributes are represented as maps defined on these sets. In LITTLE, such domains with attributes are conveniently mapped into tables of fixed-length records. Auxiliary tables are needed to accommodate arbitrary-length and multivalued attributes.

Let us consider in more detail one of these domains, *instructions*. Attributes defined on instructions include *opcode*, the operation performed by the instruction, and *arguments*, a variable-length list of arguments. Instructions in each basic block are threaded into a linked list to facilitate ordered iteration and instruction insertion and deletion. There are therefore additional attributes *next_inst*, from instructions to instructions, and *first_inst*, from blocks to instructions.

These maps are declared in **repr** statements as follows:

```
mode instruction:          elmt instructions;
  first_inst: local smap (block) instruction;
  opcode:     local smap (instruction) elmt opcodes;
  arguments:  local smap (instruction) tuple(elmt symbols);
  next_inst:  local smap (instruction) instruction;
```

In LITTLE, this collection of attributes is grouped into a table of fixed-size records, whose structure is defined as follows. If the range of a single-valued map attribute is of fixed length, the range value is a field in the record. For arbitrary-length range values, an auxiliary table is defined, and the instruction record contains fields storing the offset and length of the

attribute (LITTLE does not have pointers) in the auxiliary table. From the **reprs** above, we define, in addition to the instruction table, an auxiliary table, *argument_table*, and the instruction table fields:

opcode:	integer
arguments:	pointer to argument_table
num_arguments:	number of arguments in argument_table
next_inst:	pointer to instructions table

With these definitions, translation of typical code-manipulating global tables is relatively straightforward. For example, SETL code to iterate through all arguments of an instruction is as follows:

```
loop forall a in arguments(i) do
                . . .
end forall;
```

The corresponding LITTLE code is:

```
off = arguments(i);
do j = off + 1 to off + num_arguments(i);
          a = argument_table(j);
                    . . .
end do;
```

Tables are allocated from the heap. When a table is full, it is reallocated and copied. Some tables are only needed for certain phases and so can be freed after they are no longer used. There are 33 global tables in all.

11.5 BIT-STRING REPRESENTATION OF DATA-FLOW MAPS

To illustrate more sophisticated uses of SETL, and the use of DRSL to guide translation into LITTLE, we consider the data flow analysis module, which is a general-purpose package called from various phases of the optimizer to solve data flow problems of the bit-vectoring class [Schwartz and Sharir, 1979]. These are the simplest data flow problems that arise in global program optimization, such as live-dead analysis and redundant expression elimination. We first give some background.

11.5.1 Bit-Vector Data Flow Problems

Bit-vector data flow problems are described in a data flow framework (L, F) (cf. [Hecht, 1977; Aho and Ullman, 1977]), for which there exists a finite set

E such that $L = 2^E$, and for each f in F, there exist two subsets S_f and T_f of E such that, for each x in L,

$$f(x) = (S_f \cap x) \cup T_f. \tag{11.1}$$

Heuristically, elements of L represent (Boolean) attribute values (such as availability of expressions, live status of variables, etc.) to be computed at certain program points, and elements of F represent transformations of these values effected by program execution. Elements of F are also called *data flow maps*, and are denoted by tuples $[S_f, T_f]$.

As an example, consider the problem of detecting expression availability. The expressions that are available on exit from a block B are precisely the expressions that are available on entry to B and are not killed in B, plus those expressions that are made available within B itself. Thus we obtain a specific instance of Eq. (11.1):

$$\text{avail}_B(x) = (\text{NOKILL}_{\text{avail}_B} \cap x) \cup \text{GEN}_{\text{avail}_B} \tag{11.2}$$

where x is the set of expressions available on entry to B. These maps must be calculated for all program blocks.

Solving for x, the block entry information, for each basic block involves propagating information globally through the program, in a complex specified manner, which we will not describe more than by the following remarks. Let x_{Bn} denote the information available at entry to block B_n, and x_0 denote the worst-case assumptions upon program entry. Then

$$x_{B0} = x_0 \tag{11.3}$$

where B_0 is the entry block of the program, and

$$x_{Bn} = \cap \{ f_B(x_B) : B \in \text{predecessor blocks of } B_n \} \tag{11.4}$$

for each block B_n. The basic data flow problem is to determine the maximal fixpoint of Eqs. (11.1), (11.3), and (11.4).

SETL procedures from the data flow package implement manipulation of data flow maps in a concise fashion. For example, the procedure *meet* intersects two data flow maps; this operation is used in finding the fixpoint solution to Eqs. (11.3) and (11.4):

```
proc meet(f, g);
return
        if undefined(f) then  g
        elseif undefined(g) then  f
```

```
    else
       [f(1) * g(1), f(2) * g(2)]
    end;
    end proc meet;
```

Here f and g are two-tuples of sets, and the operator "*" is set intersection.

For each particular data flow problem, the data flow solver receives as actual parameter a large map F, which takes each edge (m, n) in the program flow graph into its corresponding data flow map $[f_{m,n}, g_{m,n}]$.

11.5.2 Data Flow Maps in SETL and LITTLE

Elements of L represent Boolean attribute values and so can be represented by bit strings. The data flow maps of the form $[S_f, T_f]$ belonging to F can be represented as pairs of bit strings, and set union and set intersection can be performed using bit-vector *and* and *or* operations. In SETL, a bit-string implementation can be obtained for a set simply by adding DRSL declarations, without modifying any of the body of the code. The LITTLE version follows closely the DRSL formulation but also optimizes the use of space in a way not achievable within SETL.

Let *dom* denote the base domain for the data flow problem, (e.g., expressions, variables, occurrences). The **mode** declarations:

```
mode data_flow_set: remote set(elmt dom);
mode data_flow_map: tuple(data_flow_set, data_flow_set);
```

define a data flow set as a bit string, and a data flow map as a two-tuple of bit strings; each bit position in a data flow set corresponds to an element in the base *dom*, and indicates membership of that corresponding element in the data flow set. To represent nodes and edges in the data flow graph, we define the bases

```
base data_flow_nodes: elmt blocks;
base data_flow_edges: tuple(elmt data_flow_nodes,
                            elmt data_flow_nodes);
```

Then the map F, passed to the data flow module, is declared as

```
F: remote smap (elmt data_flow_edges) data_flow_map;
```

that is, a single-valued map from edges to data flow maps. Since it is defined as *remote*, the range of the map is stored as a tuple: in this case a tuple where each entry is a pair of bit strings (Fig. 11.4a).

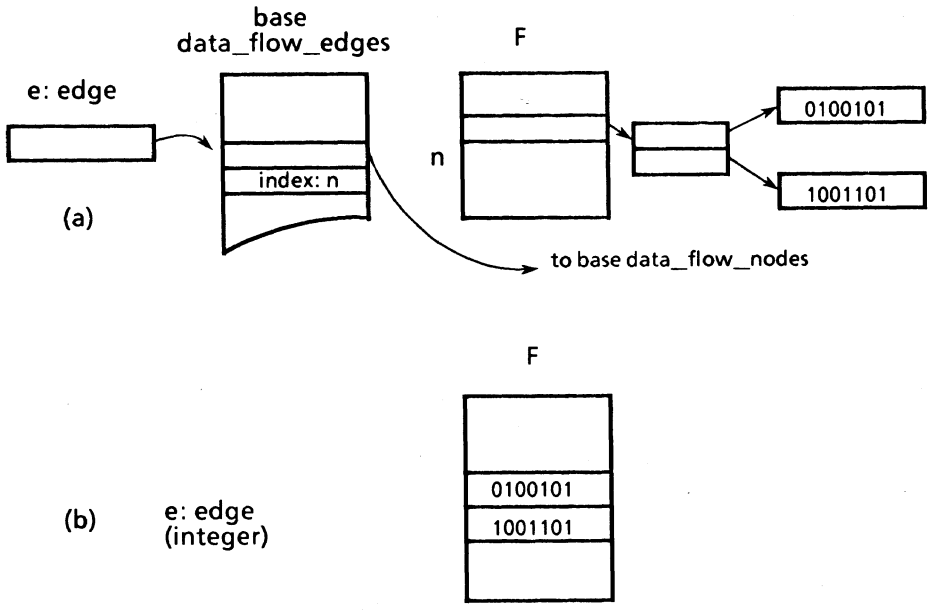
Using the SETL code as a model, the LITTLE version allocates (and later frees) a single block for F from the heap for each data flow problem. F is an

array of data flow map entries, where each entry is a pair of bit strings. The dimension of the array is the number of edges; the size of each entry in *F* is determined by the size of the domain for the problem. Because the size of the problem is known at time of allocation, we are able to obtain a more compact representation in LITTLE (Fig. 11.4b) than in SETL, where the structure for *F* has more indirection.

To illustrate the translation, the code below is the LITTLE version of the SETL *meet* procedure. In this subroutine, *andb* and *copyb* are macros that perform bit-string intersect and copy operations on arbitrary-length bit strings, where the length is passed as the last argument. The result is stored in the parameter *r*. The variable *df_map_sz* is a global variable that stores the size of the domain of each data flow map for the current data flow problem.

```
subr meet (f, g, r);
          /* f, g are pointers to entries in the array F.
             r, points to storage for the result data
             flow map. */
```

FIGURE 11.4
DATA FLOW MAPS *F*. (A) SETL DATA STRUCTURES (B) LITTLE DATA STRUCTURES



```
if (undefined(f)) then
      copyb(g, r, df__map_sz); /* Copy bit string g
                                  into result r. */

elseif (undefined(g)) then
      copyb(f, r, df__map_sz); /* Copy bit string f
                                  into result r. */
else
   /* Intersect f and g into r by anding
      bit strings f and g. */
   andb(f, g, r, df__map_sz);
end if;

end subr;
```

11.6 REPRESENTATION OF RECURSIVE TYPES

As a final example, we consider the representation of data types in the type finder module. Translation into LITTLE is more difficult here than in the previous examples. There are no **repr** declarations to rely on, since, in fact, the DRSL is not powerful enough to describe the SETL representation of types in a useful way.

The type finder attempts to determine the type of each variable in a SETL source program (see [Tenenbaum, 1974]). Since SETL is weakly typed, a single variable may denote objects of different types at different times. Therefore a type descriptor must be able to represent the union of more than one type (e.g., integer or real). Type descriptors can also be recursive, so they must be able to represent the types of the components of composite objects.

In the SETL optimizer, types are represented by tuples of the form [*type, component_type*]. These descriptors form a lattice, in which vagueness increases towards the top. The basic points on the lattice correspond to the standard types such as integer, real, and Boolean. The set of descriptors with a given basic type form a sublattice. The lattice is infinite; however, by enforcing a nesting limit on descriptors, and by treating all known-length tuples beyond a certain length as unknown-length tuples, we make the lattice finite.

The basic types are somewhere near the bottom of the lattice. Union types are somewhere near the top. The maximal type, called *type_gen*, corresponds to the union of all types. The first component of the *type_gen* descriptor is the set of all other basic types. The minimal type is called *type_zero*.

In SETL, the basic types are represented by the constants *t_om* (undefined), *t_int, t_real, t_string, t_atom, t_error, t_tuple*, and *t_set*. Common type descriptors are defined in terms of these, as in the following:

1. Integer descriptor: basic type *integer*, with no components.

```
type_int  =  [ {t_int} ];
```

2. Set descriptor: basic type *set*, with component type *type_gen*.

```
type_set = [ {t_set}, type_gen];
```

3. Defined type descriptor: the union of all defined types, with component type *type_gen*.

```
type_notom = [ {t_int, t_real, t_string, t_atom,
                                  t_tuple, t_set}, type_gen];
```

4. Pair descriptor: a tuple of length 2, with component types defined.

```
type_pair = [{t_tuple}, [type_notom, type_notom]];
```

5. Map descriptor: a set of pairs.

```
type_map = [ {t_set}, type_pair ];
```

Note that the descriptor for *type_notom* is a union type, and the descriptors for *type_pair* and *type_map* are nested. There are no **repr** declarations for such type descriptors because it is not possible in the DRSL to describe a recursive structure, that is, a composite structure with a component of the same structure. Similarly, the DRSL does not support union types. This is a significant deficiency of the current DRSL (see [Weiss and Schonberg, 1986]).

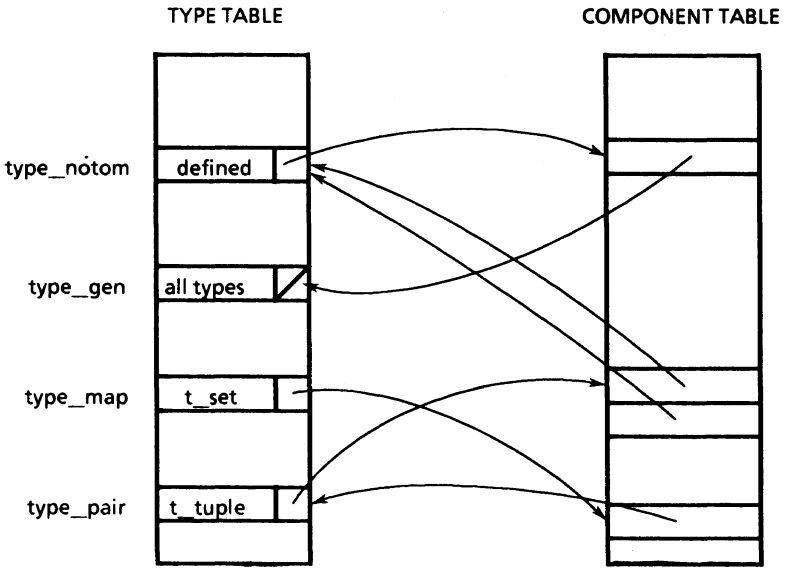
In LITTLE, descriptors are stored in a *type* table and components of descriptors in an auxiliary *component* table; for known-length tuple types there may be more than one component per descriptor. Entries in the component table point back to entries in the type table. The basic type of the descriptor, which may be a union of several types, is represented by a field of one-bit flags in a type table record. Basic types can therefore be unioned and intersected conveniently with bit string *or* and *and* operations.

Fields in the type table include the following:

basic_type	a bit string of (union of) basic types.
component_type	pointer to the component table.
num_components	number of component elements.
is_known_length	flag for known-length tuples.

To avoid generating the same type descriptor more than once, as is done in the high-level SETL specification when, for example, more than

FIGURE 11.5
LITTLE REPRESENTATION OF *TYPE_MAP*



one variable are of the same type, the type table is a hash table. Note that when comparing a new composite type with an existing table entry, it is only necessary to compare nonrecursively the outermost layer of the type (i.e., basic type and component type); components can be compared by pointer comparison because of the uniqueness of type descriptors. Figure 11.5 shows the LITTLE representation of *type_map* as defined earlier.

Translating SETL code of the type finder into LITTLE proved to be more challenging than the previous examples, in part because many of the SETL type finder routines are recursive, and also because the descriptors in LITTLE are harder to manipulate. The result is a significant code expansion. Nonetheless, once appropriate low-level data structures were chosen, the translation proceeded in a stereotyped fashion.

11.7 CONCLUSION

For large software projects, we advocate stepwise refinement, using SETL for high-level prototyping and algorithm specification. Once the algorithms are proved to be effective by experimenting with small examples, the data structure representation language may be used to further experiment with

data structures; the DRSL then aids reimplementation in a low-level language. The final product is more reliable than if it were written initially in a low-level language, since it is harder to modify and experiment with the detailed code, and the more abstract specification, substantially tested, provides a scaffolding for structuring the implementation. The SETL algorithms serve as documentation and to communicate new research results.

The translation discussed here took one year, and the LITTLE version is 35,000 lines of code, for a code size expansion factor of 2.3. The largest program optimized is 836 lines of SETL, and it took 162 seconds of execution time. The LITTLE version is faster by a factor of between 10 and 12. Unfortunately, the LITTLE version, while it can analyze programs more than twice as large as the SETL version can, is still limited in the size of programs that it can handle. It is not possible, for instance, to analyze the optimizer. This would require that the algorithms be modified to be able to incrementally analyze programs.

One might argue that if the DRSL and the optimizer were really successful, there would never again be a need for manually translating a SETL program, in that sufficient speedups of SETL programs would be gained via a combination of these tools. Automatic techniques, however, have not yet achieved the efficiency improvements that a programmer can provide. Even the data representation for type descriptors implemented in LITTLE as in Section 11.6 cannot be handled in the current DRSL nor discovered by the optimizer, although this problem has been studied [Weiss *et al.*, 1986]. The need for manual translation will be with us for a long time. The ability to reuse design, as well as actual code, can alleviate this burden.

Acknowledgment

The work of the last three authors has been supported in part by Office of Naval Research Grant N00014-82-K-0381, by grants from the Digital Equipment Corporation and the IBM Corporation, and by National Science Foundation CER Grant No. NSF-DCR-83-20085.

References

Aho, A. V., and Ullman, J. D. *Principles of Compiler Design*. Reading, Mass.: Addison-Wesley, 1977.

Dewar, R.; Grand, A.; Liu, Y.; Schonberg, E.; and Schwartz, J. T. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems* 1(1): 27, 1979.

Freudenberger, S. M. On the use of global optimization algorithms for the detection of semantics programming errors. Ph.D. thesis, Department of Computer Science, Courant Institute, NYU, 1984.

Freudenberger, S. M.; Schwartz, J. T.; and Sharir, M. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1): 26, 1983.

Hecht, M. S. *Flow Analysis of Computer Programs*. New York: Elsevier North-Holland, 1977.

Schonberg, E.; Schwartz, J. T.; and Sharir, M. Automatic data structure selection in SETL. *Sixth ACM Symposium on Principles of Programming Languages*, Jan. 1979.

Schonberg, E., and Shields, D. From prototype to efficient implementation: A case study using SETL and C. *Proceedings 19th Hawaii International Conference on Systems Sciences*, Honolulu, Jan. 1986.

Schwartz, J. T., and Sharir, M. A design for optimizations of the bitvectoring class. Courant Institute Computer Science Report #NSO-17, 1979.

Schwartz, J. T.; Dewar, R.; Dubinski, E.; and Schonberg, E. *Programming with Sets: An Introduction to SETL*. New York: Springer-Verlag, 1986.

Tenenbaum, A. M. Type determination for very high level languages. Courant Institute Computer Science Report #NSO-3, 1974.

Weiss, G., and Schonberg, E. Typefinding recursive structures: A dataflow analysis in the presence of infinite type sets. *Proceedings IEEE Computer Society International Conference on Computer Languages*, Oct. 1986.