Data Descriptors: A Compile-Time Model of Data and Addressing

R. C. HOLT
University of Toronto

Data descriptors, which have evolved from Wilcox's value descriptors [16], are a notation for representing run-time data objects at compile time. One of the principal reasons for developing this notation was to aid in the rapid construction of code generators, especially for new microprocessors. Each data descriptor contains a base, a displacement, and a level of indirection. For example, a variable x lying at displacement 28 from base register B3 is represented by this data descriptor: @B3.28. The general form of a data descriptor is $@^k b.d.i$ where k gives the number of levels of indirection, b is a base, d is a displacement, and i is an index.

Data descriptors are convenient for representing addressing in Fortran (with static allocation and common blocks), in Pascal and Turing (with automatic allocation and stack frames), and in more general languages such as Euclid and PL/I. This generality of data descriptors allows code generation to be largely independent of the source language.

Data descriptors are able to encode the addressing modes of typical computer architectures such as the IBM 360 and the PDP-11. This generality of data descriptors allows code generation to be largely machine independent.

This paper gives a machine independent method of storage allocation that uses data descriptors. Techniques are given for local optimization of basic arithmetic and addressing code using data descriptors. Target machine dependencies are isolated so that the part of the code generator that handles high-level addressing (such as subscripting) is machine independent. The techniques described in this paper have proven effective in the rapid development of a number of production code generators.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation*

General Terms: Languages

Additional Key Words and Phrases: Addressability, addressing modes, array subscripting, base displacement addressing, code generation, code optimization, compiler structure, compilers, data alignment, data descriptor, display based addressing, language translators, machine idioms, machine-independent code generation, optimal addition, portable compiler, storage allocation

1. INTRODUCTION

Data descriptors are a notation for representing run-time objects such as variables at compile time. The objects being represented may be simple, such as the integer value 13, or relatively complex, such as a FORTRAN common block. Data descriptors can represent addressing in languages such as FORTRAN, Pascal,

The research reported here has been supported in part by the Natural Sciences and Engineering Research Council of Canada and by Bell Northern Research Ltd.

Author's address: Computer Systems Research Group, University of Toronto, 10 King's College Road, Toronto, Ont., Canada, M5S 1A4.

PL/I, and Ada. This generality allows one to write major portions of a code generator that are independent of the source language.

Data descriptors can be mapped to the addressing modes in traditional computer architectures such as the IBM 360 or the PDP-11, as well as those of the less conventional new microprocessors such as the MC68000. Because of this, data descriptors allow one to write major portions of a code generator that are independent of the target machine.

1.1 Background

The author was one of the implementors of the PL/C compiler for PL/I [4], for which Wilcox developed *value descriptors* [16]. Data descriptors, and many of the ideas presented here, are based on experience in the PL/C project and on Wilcox's value descriptors. Data descriptors were developed during the design of the Toronto Euclid compiler [5], to serve as the basis of highly retargetable code generators; this compiler generated the PDP-11 code. Barry Spinney [14] extended these techniques to develop the code generator for Concurrent Euclid [3], targetted originally for the MC68000. This code generator has been retargetted for the MC6809, PDP-11, VAX, Intel 432, IBM 370, and Intel 8088/8086. The same techniques have been used in the Turing compiler [7] to generate code for the VAX, 370, and 8088/8086. One of these compilers can be retargetted to a new architecture in roughly two man-months. These code generators produce good code, similar to that produced by other production compilers, such as the Berkeley Unix C compiler. The rest of this paper presents data descriptors and the ways they are used in allocating storage and generation code.

1.2 Organization of Paper

The remainder of this part of the paper (Part 1) introduces data descriptors by giving examples and by presenting a formal definition. Part 2 shows how data descriptors can represent (1) software objects, such as records in a high-level language, and (2) hardware addressing modes, such as displacements from base registers. Part 3 gives a machine-independent storage allocation algorithm based on data descriptors, which can be used for Pascal-like languages. Part 4 shows how data descriptors support the local optimization of addition, subtraction, and multiplication, as well as optimization by selection of specialized machine instructions. Part 5 gives machine-independent code generation techniques for high-level addressing operations, such as subscripting and use of a display.

1.3 Some Examples

We will give some simple examples of data descriptors, then a precise definition of data descriptors, and more detailed examples.

A data descriptor specifies a run-time object by giving its base, displacement, and index, in units of bytes or words. In many cases the index is omitted. The notation can be expanded to specify other attributes such as size and alignment, but we will ignore such possibilities in this paper. As an example, variable x, located 28 bytes beyond base register B3, would be represented as @B3.28. We will use "bytes" as our unit of storage; but another unit, such as words, can be used instead). The @ sign means we *fetch* the value located 28 bytes beyond the

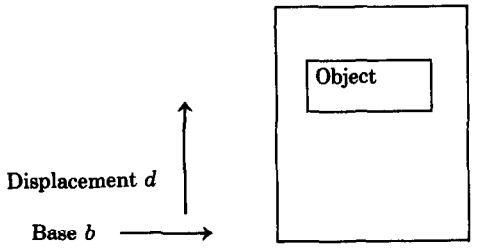


Fig. 1. Base-displacement addressing model.

position located by register B3. The address of x would be written as @B3.28, or simply B3.28. If x is a pointer, then the value pointed to by x is @@B3.28 or $@^2$B3.28.

The displacement in a data descriptor is a manifest value (a value known at compile time). It can be negative as well as positive. For example, the constant value −17 is represented by the data descriptor $@^0$null.−17 or simply null.−17, or more simply −17.

The base of the data descriptor can be a machine register, such as one of the registers of an IBM 360, or it can be null. It can also be of a more general form; for ALGOL-like languages, it is convenient to allow the base to be a "lexical level." For example, variable y, whose declaration is nested inside 3 scopes in Pascal, can be represented as @L3.18 where y lies 18 bytes into the activation record for lexical level 3.

For data descriptors of the form $@b.d$, the base b locates a *data area*, and d gives a displacement into the area; this model is called base-displacement addressing. This is illustrated in Figure 1. The object $@b.d$ is a value in the data area. If we remove the @ sign (the descriptor becomes $b.d$), we get a different object, namely the address of our previous object. If we put on another @ sign (the descriptor becomes $@@b.d$), we get another object, namely, the value pointed to by $@b.d$. As we will see, the base b is itself an object (a pointer value) and may be represented by a data descriptor.

As this example illustrates, we assume that there is a linearly addressed memory, whose units are bytes. We can fetch values from this memory; we sometimes consider a fetched value to be the address of another value in the memory.

1.4 Formal Definition of Data Descriptors

We now give the formal definition of data descriptors.

Definition 1. A data descriptor is $@^k b.d.i$ where

k = number of levels of indirection ($k \geq 0$)
b = base
d = displacement (an integer constant)
i = index

The base b is null, a machine register Rn or a lexical level Ln. If b is Ln, then there is another data descriptor which gives the value of Ln. The index is null or is a machine register Im. For the purposes of this paper, register Rn should be

considered to be a physical machine register; however, R*n* could alternatively be considered to be a logical register that is allocated a physical register at some intermediate stage of code generation.

To shorten the notation, we commonly omit $@^k$ when $k = 0$, the base or index when null, and d when $d = 0$. (Of course, we do not omit all portions of the data descriptor $@^0$null.0.null. We write it as 0.)

The "value" of a data descriptor is the sum of its base, displacement, and index all taken k levels indirect. We define this notion formally in two steps. First, we define the "numeral" of a data descriptor as the sum of its base, displacement, and index. Then we use the concept of a numeral to define the descriptor's "value."

Definition 2. The *numeral of* $@^k b.d.i$ is the sum of (1), (2), and (3):

(1) integer value d,

(2) if b is null then zero,
if b is machine register R*n* then the contents of R*n*,
if b is lexical level L*n* then the value of the data descriptor associated with L*n*,

(3) if i is null then zero,
if i is machine register I*m* then the contents of I*m*.

Notice that the definition of "numeral" does not depend on the number of levels of indirection.

Definition 3. Let N be the numeral of $@^k b.d.i$. The *value* V of $@^k b.d.i$ is defined as

if $k = 0$ then $V = N$
else V is the object in memory pointed to by the value of $@^{k-1} b.d.i$.

The definitions of "numeral" and "value" are recursive in that the data descriptor of a lexical base has a "value," defined in terms of its "numeral," defined in terms of "value," etc. In other words, the base of a data descriptor may be specified by another data descriptor. This recursion stops at a data descriptor whose base is a machine register or is null.

2. REPRESENTING HIGH-LEVEL OBJECTS AND ADDRESSING MODES

Next, we show how data descriptors can conveniently represent both software objects, such as Pascal variables, and hardware addressing modes.

2.1 Representing Objects in a Pascal Program

The use of data descriptors will be demonstrated in terms of an example Pascal program. Figure 2 shows declarations at three lexical levels, namely, inside (1) program P, (2) procedure Q, and (3) record R.

Given that variables w and x in P are at lexical level 1, their base is L1. Since w is the first variable at this lexical level, its displacement has been set to zero. Assuming that w occupies 2 bytes, it follows that x's displacement is 2. Parameters a and b are at lexical level 2, with base L2. Parameter a is a "value" parameter and behaves like an initialized local variable. Parameter b is a "var" or reference

```
program P(output);
  var w: integer;                    @L1.0
  var x: char;                       @L1.2
  procedure Q(
    a: integer;                      @L2.0
    var b: char);                    @@L2.2
    const c = 13;                    @^0null.13 = 13
    type R =
    record
      f: real;                       @L3.0
      g: char                        @L3.4
    end;
    var y: R;                        @L2.8
      z: integer;                    @^0R3.0 = R3
    begin{Q's body} end;
  begin{P's body} end.
```

Fig. 2. Example Pascal program.

parameter whose value is found by following the pointer @L2.2 and, hence, the value of *b* is represented by @@L2.2. The constant *c* is represented with zero indirection and a null base, namely as @^{0}null.13. A non-numeric constant, such as a character, is represented by its ordinal.

The bases for lexical levels 1 and 2 may be represented by machine registers or some other appropriate mechanism such as a memory resident display. Whatever this mechanism is, it is assumed to be specified by data descriptors associated with L1 and L2.

Fields *f* and *g* in record R are at lexical level 3. The base of R depends on the particular instance (variable) of type R. For example, in *y.g* the base of the record is the location of variable *y*. This means that any data descriptor associated with L3 is ignored because it is replaced by the location of the variable preceding the dot (the location of *y*, for example). (If R were a Euclid module then L3's data descriptor would specify the module's data, which is accessible whenever execution occurs inside R [8]).

As can be seen from these examples, data descriptors can be thought of as a generalization of lexical-level/order-number addressing [13].

Following the parameters *a* and *b* and preceding the local variables *y* and *z*, we may need to leave space for control information such as the return address and linkage. Variable *y* is of type R. We have made its displacement 8, leaving 4 bytes for control information.

Variable *z* has been allocated register R3, instead of a memory location. Hence, *z*'s data descriptor is @^{0}R3.0 or simply R3. Note that this is the only data descriptor in our Pascal example that specifies a machine register. If we forego the optimization of putting variables in registers, there is no need to use machine registers when giving the data descriptors for variables, constants, and fields in Pascal.

2.2 Wilcox's Value Descriptors

The preceding Pascal example will be used to illustrate the difference between data descriptors and Wilcox's value descriptors [15, 16]. Foremost, data descriptors are a *notation* intended for human manipulation, including instructional purposes. Value descriptors are a *data structure*, intended for use in a compiler

implementation; Wilcox defined value descriptors using PL/I pointers and structures. When data descriptors are encoded as data structures in a compiler, they become much like value descriptors in that both encode a displacement, base, and index.

Data descriptors generalize value descriptors by introducing explicit levels of indirection, specified by @ signs. Value descriptors have exactly one level of indirection, except for the special case of constants, such as 13. As a result, a value descriptor can represent only those data descriptors having the forms d (a constant) or $@b.d.i$ (a value in memory). A value descriptor cannot represent various useful forms such as L2.8 (the address of record y) or @@L2.2 (used above for reference parameter b). (However, it is possible to represent @@L2.8 by a *pair* of value descriptors, the first representing @L2.8 and the second designating the first as its base.)

Value descriptors do not allow null indirection. When a data descriptor has null indirection, its value is the sum of its base, displacement, and index. This form is important because it is the basis of the optimal addition algorithm given later in the paper, as well as being used in the algorithms for subtraction, multiplication, and high-level addressing.

The discussions given in the present paper on representation of high-level objects and machine-addressing modes, on generating machine idioms, on resolving lexic basis, and on forcing addressability parallel discussions by Wilcox on these same topics. The present paper introduces methods of using data descriptors for storage allocation, for optimal addition (and other operations), and for generating code for high-level addressing.

2.3 Representing Other Objects in High-Level Languages

This section explains how data descriptors can represent various objects in high-level languages, namely: variables at absolute addresses, pointers, the address operator, "own" variables, common blocks, and external variables.

In languages such as Euclid, variables can be specified to be allocated at absolute locations. For example,

```
var ttyBuffer(at 177562#8): char;
```

creates the variable ttyBuffer with octal address 177562. The data descriptor for ttyBuffer is @null.177562#8; note that the null base and single level of indirection specify that ttyBuffer has an absolute address.

Pascal has pointer variables; the "up-arrow" operator accesses values located by pointer variables. If p is a pointer with data descriptor D, then $p\uparrow$ has the data descriptor @D. Thus the source level up-arrow operator is equivalent to "@".

Sometimes it is necessary to access the address of a variable. For example, when variable x is an actual parameter for a Pascal "var" parameter, the address of x must be passed to the subroutine. As another example, the C language has the "address" operator &, so $\&x$ evaluates to the address of x. For an object represented by data descriptor D, we denote its address as $@^{-1}$D. For example, if x is $@^1b.d$, then the address of x is $@^{-1}(@^1b.d)$ or simply $@^0b.d$, or more simply $b.d$. Thus the & operator of C is equivalent to $@^{-1}$.

When a data descriptor has zero indirection, as in $@^0b.d = b.d$, we say it is a *direct* data descriptor. It is problematic to try to determine the address of a direct

data descriptor; for example, there is no unique address for a direct value such as 9. In Pascal, this difficulty never arises because there is no address operator and because "var" actual parameters must be variables (not constants or expressions). In PL/I, where constants and expressions can be passed to reference parameters, the difficulty is solved by implicitly creating "dummy arguments." A dummy argument is a nameless variable initialized to the value of the actual parameter; the address of this implicit variable is passed to the subroutine.

ALGOL-60 has the concept of "own" variables, whose values persist from activation to activation of the containing block. In PL/I terminology, these variables are allocated "statically." We can invent a dummy lexical level, call it L0, for a data area in which "own" variables are allocated. With this artifice, "own" variables require no special consideration because they are represented by ordinary data descriptors.

FORTRAN subroutines can share variables that lie in "common blocks." Each common block is a data area. A particular subroutine accesses a variable in a common block by knowing the base of the block and the displacement of the variable within the block. If the base of the common block is *b* and the displacement is *d*, then the variable is represented by data descriptor *@b.d*. In typical implementations of FORTRAN, the base *b* is a machine register or a pointer in memory. We can give each common block a number and consider that common block number *n* has base L*n* (lexical level *n*). With this convention, our data descriptors can handle common blocks. If this use of "lexical levels" to represent common blocks is considered too artificial, one can extend data descriptors so that the base is allowed to be C*j* (common block *j*).

Languages like PL/I and C provide "external" variables, which can be shared among different compilations by giving their names. Each external variable can be treated as a (small) common block. Alternatively, each can be treated as a reference parameter, whose value is found via an implicit pointer.

2.4 Representing Addressing Modes in Machine Instructions

We have shown how data descriptors can represent objects in high-level languages. We will now show how addressing in instructions of typical computer architectures can be modelled by data descriptors. We have picked the IBM 360 and the PDP-11 as example architectures; these architectures provide addressing modes similar to a large number of other architectures, including: NS32000, VAX, Zilog 8000, Intel 8086, Motorola 6809, and Motorola 68000. We will first give a brief description of the addressing modes of these two architectures.

The IBM 360 has the following addressing modes:

- I Immediate mode. The operand is a constant (necessarily a byte).
- R Register mode. The operand is contained in a machine register.
- S Storage mode, written *d*(*b*). The operand is located by adding a 12-bit displacement to a register.
- X Index mode, written *d*(*b*, *i*). Like S mode, but a second register (the index) is included in the address calculation.

In the 360, the operation determines the addressing modes of the operands. For example, the AR (Add Register) instruction adds operands both with R mode, while the A (Add) instruction adds an X mode operand to an R mode operand.

By contrast, on the PDP-11 the addressing mode is specified by the operand itself (not by the operation) and all modes are allowed for most operations. Here is a list of the basic addressing modes of a PDP-11:

#d	Immediate mode. The operand is constant value *d* (either a byte or a 16-bit word).
@#d	Absolute mode. The operand lies at address *d*.
r	Register mode. The operand is contained in a machine register.
@r	Register deferred mode. The register's contents point to the operand value.
d(r)	Index mode. The operand is located by adding a 16-bit displacement to a register. This is similar to the 360 S mode.
@d(r)	Index deferred mode. This is like the preceding mode *d(r)* except there is another level of indirection.

We do not include the PDP-11 auto increment/decrement modes in this list because they imply a side-effect (changing a register), rather than simply specifying a value. Auto increment/decrement modes are useful when comparing/moving nonscalar items and when allocating/deallocating space on the run-time stack. These special situations can be handled using data descriptors by temporarily tagging the descriptor with an auto increment/decrement flag when the side-effect is desired.

Table I gives data descriptors with corresponding addressing modes of the 360 and the PDP-11. The items in square brackets can be used to provide the desired addressing. The dashes indicate that the architecture does not support the particular mode.

This table shows that the designers of these two architectures have chosen to favor certain modes at the expense of others. For example, the 360 supports an index register, but has no double indirect (@@) addressing modes. The 11 has one double indirect mode (*@@r.d*), but does not allow an index register. (Technically, the 11 also supports the *@@r* mode, but only when auto increment/decrement is specified.) Those modes that a particular architecture supports are called *immediately addressable* [16]. Notably absent in both architectures are the "effective address" modes *r.d* and *r.d.i*. Presumably, the designers of these architectures have favored certain modes based on an expectation of the frequency of their use.

The 360 uses a particular register number (zero) to specify the null register; for example, the X-mode operand *d*(0, 0) is equivalent to *@d*. Similarly the X-mode operands *d*(0, *r*) and *d*(*r*, 0) are equivalent to *@r.d*. The 360 operands 0(*r*, 0), 0(0, *r*), and 0(*r*) are equivalent to *@r*.

The 360 has a special instruction called Load Address (LA) that effectively decreases the indirection of its X-mode operand. For example, "LA R3,2(R1, R2)" loads register R3 with the sum of 2 and the contents of registers R1 and R2. By special case analysis, the LA operation can be pressed into action to support the *r.d* and the *r.d.i* modes. There is unfortunately a severe shortcoming of LA, namely, the sum it produces is truncated on the left to a 24-bit unsigned number, because 360 addresses are 24 bits long. Hence LA is useful only when it is known that the result is a sum lying in the range 0 to $2^{24} - 1$ or is an address.

Table I. Corresponding Addressing Modes of the 360 and the PDP-11

Data descriptor	360 Mode	11 Mode
d	I	*#d*
@d	[S, X]	*@#d*
@@d	—	—
r.d	—	—
@r.d	S[X]	*d(r)*
@@r.d	—	*@d(r)*
r	R	*r*
@r	[S, X]	*@r*
@@r	—	*[@0(r)]*
r.d.i	—	—
@r.d.i	X	—
@@r.d.i	—	—

The PDP-11 provides modes that specify that there is no base register or that the displacement is zero. For example, the 11 modes *@r* and *0(r)* describe the same operand, but *@r* is preferable because it does not require a displacement (an extra 16 bits). The 11 does not directly support *@@r*, but the form *@0(r)* specifies the same value using a 16-bit displacement equal to zero.

When an addressing mode is missing from an instruction repertoire, it can be simulated by generating extra instructions. For example, the mode *r.d* can be simulated by explicit addition of *r* and *d*. On the PDP-11, this addition must be generated by a compiler to compute the address *r.d* of variable *@r.d* so the variable can be passed to a reference parameter. On the 360, the mode *@@r.d* is missing, so it is simulated by explicit code for the second level of indirection. This code is generated by a compiler for accesses to reference parameters.

3. STORAGE ALLOCATION

We have shown how data in high-level languages can be represented in a machine-independent way by data descriptors. This suggests that the space for data can be allocated in a machine-independent way. This is feasible for Pascal-like languages, and is done in the Concurrent Euclid compiler [10] and the Turing compiler [6].

3.1 Machine-Independent Algorithm for Storage Allocation

We will give an allocation algorithm that computes a data descriptor for each variable, parameter, or record field. For each of these, it determines *k*, *n*, and *d* in $@^k Ln.d$. Also computed are the size and alignment of each activation record and user-defined type.

We first give the basic technique that is parameterized by the sizes and alignments of simple types, and then we suggest extensions to handle various target machine peculiarities.

We assume that each simple type of the source language has a known size and alignment. These sizes and alignments are machine-dependent parameters used by the machine-independent allocator. For Pascal on the PDP-11 we need a table

Table II. Type Size and Alignment

Simple type	Size	Alignment
char	1	byte
Boolean	1	byte
integer	2	word (double byte)
float	4	word (double byte)
address	2	word (double byte)

such as in Table II. The table should be extended if there are more simple types, such as "long integer" or "float double." For Pascal we need to parameterize the computation of the size and alignment of "sets," for example by making all sets words or by allocating a sufficient number of bytes to hold the bits of the set. We also need to parameterize the size and alignment of Pascal's subrange types.

Some computers, such as the Motorola 6809, require no alignment; we ignore alignments when this is the case.

Our allocation technique depends on the following observation. Other than simple types, the only data objects that need to be allocated are arrays and records. (We are ignoring the allocation of space for temporaries.) We generalize the concept of a record to include not only those explicitly declared by the user via the construct *record ... end*, but also the data areas (activation records) of subroutines. For example, in Figure 2, the activation record for program P has two data fields, w and x.

For each record (implicit or explicit), the allocator determines the lexical level Ln, which is used as the base of data declared in the record. This is done by keeping a count of the number n of surrounding records. The allocator will not concern itself with how the base of a record is implemented at run-time. The question of this implementation is left to a successive phase of the compiler, which can use the lexical level n to look up the appropriate base in a data structure called a "compile time display"; this is explained in a following section.

Each field of a record is allocated by determining its level of indirection k, its base Ln, and its displacement. We determine n as simply the current static depth of nesting. For local variables and fields of user records, we allocate space according to the size of the variable's type and we use a single level of indirection, i.e., $k = 1$. For reference parameters, we allocate space for an address and use double indirection, i.e., $k = 2$.

The allocator needs to calculate the size and alignment of the non-simple types, namely arrays and records. The algorithm in Figure 3 determines all sizes and alignments.

In case (1) the type is simple, and "size" and "align" are looked up in initialized arrays called tableOfSize and tableOfAlign. We assume that t is an entry in a table representing types and that t.kind has already been set to represent the given simple type, i.e., set to represent char, Boolean, etc.

In case (2) the type is an array, and its size and alignment are computed from its index and element types. For arrays, we assume that type table entry t for the array has fields indexType and elementType pointing to type table entries for the element and index types.

```
(1) Simple:
        size := tableOfSize[t.kind];
        align := tableOfAlign[t.kind]
(2) Array:
        size := (t.indexType↑.upper−t.indexType↑.lower+1)*t.elementType↑.size;
        align := t.elementType↑.align
(3) Record:
        sizeSoFar := 0;
        alignSoFar := 0;
        for each field do
        begin
          displacement := RoundUp(sizeSoFar, field.align);
          sizeSoFar := displacement+field.size;
          alignSoFar := max(alignSoFar, field.align)
        end;
        size := RoundUp(sizeSoFar, alignSoFar);
        align := alignSoFar
```

Fig. 3. Machine-independent allocator.

In case (3) the type is a record, and its size is the sum of its field sizes, as rounded up to force required alignment. The alignment of the record is the maximum of its field alignments. The overall size and alignment are accumulated a field at a time. As each field is handled, its displacement is computed as the current size rounded up for alignment.

For each field representing a variable, its displacement d, the current lexical level Ln, and the level of indirection k are placed in the symbol table entry for the field. Taken together, these three form the data descriptor $@^k$L$n.d$ for the field. If the field represents a reference parameter, the data descriptor is given an extra level of indirection, as in $@^2$L$n.d$. The creation of this data descriptor constitutes the allocation of space for the field.

This completes the description of the basic allocation algorithm. Some points need expansion to handle a particular source language.

Notably lacking is any consideration of Pascal's variant records. These are handled by overlaying the data for each variant. The algorithm for allocating records must reset "sizeSoFar" each time a new variant is encountered to the size of the record's common data. The overall size of the record is the maximum value of sizeSoFar (rounded up) at the ends of the variants. The overall alignment is the maximum of all field alignments in the record including all variants.

We have ignored the allocation of space in an activation record to hold control information such as the return address, registers, and linkage among activation records. This part of allocation is machine dependent, but can be parameterized across a large class of architectures.

The allocator may be given a set of registers that it can allocate to scalar variables. These can be allocated in a clever way, for example, based on usage counts for the variables. They can be allocated in a naive way, as is done in the Toronto Euclid compiler, by allocating them in order of declaration as long as the registers last. In the Concurrent Euclid language, the user explicitly requests that certain variables be allocated registers; these requests are granted as long as

registers are available. In Figure 2 note that local variable z of procedure Q has been allocated a register.

3.2 Some Details of Record Allocation

For use in production compilers, certain refinements of the record allocation algorithm are required. In architectures similar to the PDP-11, arguments to a subroutine are pushed onto a run-time stack. This stack grows downward, meaning that the stack top has a smaller address as a result of each push. This implies that the displacements for successive arguments are decreasing. The record allocation algorithm must be modified to handle these backward growing displacements. Alternately, the compiler can reverse the order of arguments, as is done by the UNIX PDP-11 C compiler.

Another problem with arguments is that the target computer may create extra space on the stack as a result of pushes. For example, when a byte is pushed onto the PDP-11 stack, the stack actually grows by two bytes. The allocation algorithm must be aware of these machine idiosyncracies.

A fine point of record allocation concerns the problem of holes (unallocated space) in records created by forcing alignment of fields. A more elaborate, two-pass allocation algorithm can sort fields according to decreasing alignment requirements, thereby eliminating holes in records. While this is possible, it has the disadvantage that the user can no longer predict the order of fields in the allocated record, and hence loses the ability to match field displacements of records in existing files. An interesting feature of the algorithm as presented in Figure 3 is that all fields are given their required alignments by an algorithm that uses only one pass over the record declaration.

4. LOCAL CODE OPTIMIZATION

We have shown how to use data descriptors to perform machine-independent storage allocation. We will now show how data descriptors can make local code optimization more machine independent. We will begin by considering addition, because it is the most common explicit (user-written) arithmetic operation [1], and it is implicitly used in subscripting and field selection. Next, we will consider optimization of subtraction and multiplication, and then will show how efficient idioms (specialized machine instructions) can be easily generated.

4.1 Optimal Code for Addition

We will develop a code generation algorithm to generate optimal code for addition of a restricted form of data descriptor. This algorithm is called "super-add," because it generates the best possible code, given the constraints that are formalized below.

We will limit our attention to data descriptors of the restricted form $@^k Rn.d$, where Rn can be null, d can be zero, and k is 0 or 1. We are assuming that any lexical level base Ln has been resolved, either to machine register Rn or to null. We are also assuming that there cannot be an index register Im. (Our results can be extended to handle other cases.)

We assume that there are two classes of machine registers—those used for temporary values, denoted Ti, and those permanently assigned, denoted Bi. Our

generated code must never modify a Bi register. The super-add algorithm assumes it has enough temporary registers Ti to hold all temporary results.

Given these assumptions, we want to generate the best code for adding two data descriptors, or more precisely:

Problem. Given data descriptors D and E of the restricted form @^{k}Rn.d, specify code to add D and E that (1) is of minimal size, and (2) uses minimum temporary registers, such that the result is represented by a data descriptor of the same restricted form.

We have chosen to minimize size rather than speed, as size is easier to quantify. The reader may want to convince himself that speed as well as size is optimized for the obvious interpretation of the model we will give.

The machine model we use has two machine instructions of interest, namely,

(1) move, written $A := B$ (assign B to A)
(2) add, written $A :+ B$ (add B to A)

The machine model restricts A to be a register Rn and B to be of one of the forms:

(1) *Rn*	(a temporary Ti or a permanent Bi)
(2) *d*	(an integer constant)
(3) @Rn.d	(a value in memory)

The *size* of an instruction is one word, plus one more if there is a displacement. For example, letting t and b represent registers:

$t := b$	Assign register b to register t. Size is 1.
$t :+ b$	Add register b to register t. Size is 1.
$t :+ d$	Add displacement d to register t. Size is 2.
$t :=$ @b.d	Move value in memory to register t. Size is 2.

The machine model and its costs correspond to the add and move instructions of the PDP-11, with the target A restricted to be a register and the source B restricted to be one of the PDP-11 modes: (1) Rn, (2) #d, or (3) d(Rn).

Consider the case in which the right operand has the form @bR.dR. For this right operand form, optimal code can be listed for each possible form of left operand, as shown in Table III. The code shown for these cases can be proven to be optimal by enumerating all sequences of model instructions up to length 4 (the maximum size for any code given here).

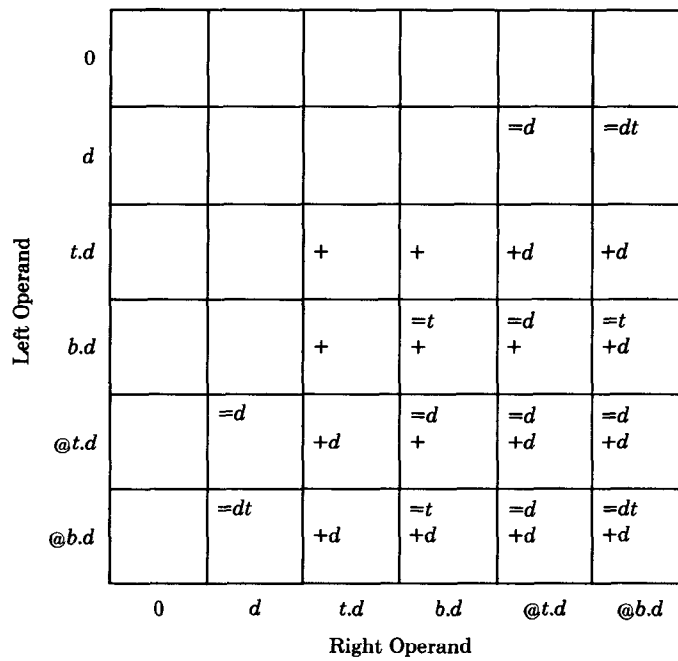
The super-add algorithm must handle 36 cases, namely, the six forms of left operands times the six forms of right operands. Rather than enumerating all these cases, we summarize the results in Figure 4. The six cases that we developed explicitly are represented by the rightmost column of this table.

For particular left and right operands, a table entry gives the instructions to be emitted, the number of displacements to be emitted, and whether a temporary register is to be allocated.

Blank table entries indicate that no code should be emitted. For example, the first column, in which the right operand is zero, never requires code to be generated. In the second column, code need not be generated in many cases; for

Table III. Optimal Code for Adding to @*b*R.*d*R

Left operand	Code	Result	Size	Temps
0	none	@*b*R.*d*R	0	0
*d*L	*t* := @*b*R.*d*R	*t*.*d*L	2	1
*t*L.*d*L	*t*L :+ @*b*R.*d*R	*t*L.*d*L	2	0
*b*L.*d*L	*t* := @*b*R.*d*R *t* :+ *b*L	*t*.*d*L	3	1
@*t*L.*d*L	*t*L := @*t*.*d*L *t*L :+ @*b*R.*d*R	*t*L	4	0
@*b*L.*d*L	*t* := @*b*R.*d*R *t* :+ @*b*L.*d*L	*t*	4	1



Notation:
- *b* Permanent register
- *t* A temporary register
- *d* Displacement
- 0 Zero
- \+ Add instruction required
- = Move instruction required

Fig. 4. Cost matrix for optimal addition of data descriptors.

example, if the left operand is of the form *t.d*, say T3.6, and the right operand is of the form *d*, say 12, then the result is T3.18 with no generated code.

The bottom entry of the second column contains the three symbols *t*, =, and *d*. This means that to add the forms @*b.d* (left operand) and *d* (right operand), a temporary (*t*) must be allocated and a move (=) instruction with a displacement (*d*) must be generated. In the case of operands @B2.8 and 11, the generated instruction would be

$$\text{T4} := \text{@B2.8}$$

where T4 is the allocated temporary register. The result of the addition is T4.11.

The case of adding @*b.d* to *b.d* requires one displacement and two instructions. The minimum cost code can be generated in one of two ways. Denoting the left operand as @*b*L.*d*L and the right as *b*R.*d*R, the generated code could be either

$$t := @bL.dL$$
$$t :+ bR$$

or

$$t := bR$$
$$t :+ @bL.dL$$

In both cases the result is *t.d*R. There are several other cases in the table where there is a choice of (optimal) code.

The algorithm for generating code can use the cost matrix to derive the required sequence of code. Alternately, the code generator could be given a table that explicitly gives the required sequences. David Wortman has shown that the optimal code to be emitted can be encoded in an extremely compact table using techniques similar to those of Lowry and Medlock [11]. In the case of the Euclid and Turing code generators, there is no explicit table. Instead, the code generator does a case analysis written in S/SL [9], based on inspection of the operands to determine code sequences to be generated.

Our machine model is simple and is not intended to correspond exactly to any production architecture. With obvious extensions, it corresponds to the PDP-11 architecture; it is with these extensions that it is used in the Euclid and Turing compilers.

For different architectures the number of relevant operand forms will vary. But the methodology of constructing the table remains the same, namely, a straightforward inspection of instructions and operands supported by the architecture. In fact, it is possible to automatically generate the table for given constraints on instructions and operands. The blank table entries remain blank for all architectures, as these entries represent compile-time implementation of addition.

4.2 Optimizing Subtraction and Multiplication

We have discussed optimization of addition in some detail because that operation occurs so often. In this section we briefly consider subtraction and multiplication. These operations are important because they occur implicitly in subscripting. The array reference $a[i]$ is represented by

$$\text{Addr}a + (i - \text{Lower}) * \text{Size}$$

where Addr*a* is the address of array *a*, Lower is the array's lower bound, and Size gives the number of storage units occupied by each element of the array.

The only case of subtraction we will consider is the subtraction of a constant d from an arbitrary left operand V, i.e., the form $V - d$. In subscripting calculations, this is the most important case because d represents the lower bound, which is usually known at compile time.

Since d is a constant, we can negate it at compile time to create c, where $c = -d$; so we can replace $V - d$ by $V + c$. This transforms the subtraction into

an addition, so we can optimize this case of subtraction using super-add, which we have already discussed.

The only case of multiplication we will consider is when an arbitrary left operand V is multiplied times a constant d, i.e., the form $V * d$. In subscripting calculations, this is the most important case because d represents the element size, which is usually known at compile time.

The following form of V is an important special case: R$n.e$, i.e., value V is the sum of constant e and possibly null register Rn. When the register is null, V is simply the constant e. This case of a constant (e) times a constant (d) is fairly common, occurring for constant subscripts, and is carried out at compile time.

The case of a non-null register in R$n.e$ is considerably more common, as we will now illustrate. Consider the Pascal statement

$$a[i] := x$$

that can be represented as

$$@(\text{Addr}a + (i - \text{Lower}) * \text{Size}) := x$$

where Addr$a$, Lower, and Size are as previously defined. Let us assume that the value of i has been placed in register Ri, and that the value of Lower is 1. The super-add algorithm will reduce the subexpression (i − Lower) to the data descriptor Ri. − 1 without generating any code. Now the subexpression (i − Lower) * Size is reduced to (Ri. − 1) * Size, and this is an instance of the form (R$n.e$) * d.

Assuming that Rn is a temporary register, the best way to implement (R$n.e$) * d is to generate multiplication of Rn by d. Then the value of (R$n.e$) * d is R$n.c$, where c is computed at compile time as $e * d$. We have distributed the multiplication of d across Rn and e to avoid generating code for the addition.

For the expression $V * d$, we have shown how to optimize the generated code for special cases of V. There are also important special cases of the constant d that occur in subscripting. Most important are the cases of d equal to 1, 2, 4, or 8. When d is the size of an array element, it commonly has one of these values, as they are the sizes of scalars such as characters, Booleans, integers, and reals for byte-oriented architectures.

When $d = 1$, the optimization of $V * d$ is obviously to generate no code giving the result V. When d is a power of 2, including 2, 4, and 8, the best code for typical architectures is to avoid a multiply instruction in favor of "shifting left."

Although they do not really deserve the names, we have called our techniques for optimizing subtraction and multiplication "super-subtract" and "super-multiply."

In a later section we will show how these optimized add, subtract, and multiply techniques allow us to specify machine-independent, high-level addressing such as subscripting.

4.3 Quality of Generated Code

The techniques just described illustrate how data descriptors support the generation of good code with modest compiler complexity. There are code generation techniques, such as those used in the FORTRAN-H [11] and BLISS-11 [18]

compilers, which can produce smaller and/or faster code. The techniques described in [18] for BLISS-11 are notable for the extensive set of optimization cases handled, including all or almost all of the optimizations described in the present paper. However, their algorithms and data structures for producing optimizations are quite different and much more elaborate than those described in the present paper. The advantage of the data descriptor approach is that it tends to lead to smaller and simpler code generators that are highly retargetable.

4.4 Generating Machine Idioms

Computer architectures contain idioms (specialized instructions) that handle special case operations. For example, the PDP-11 has special instructions called INC and DEC that add and subtract one to their operands, respectively, which are faster and smaller than corresponding add and subtract instructions.

A code generator can produce significantly better code if it can recognize patterns that correspond to machine idioms. As described by Wilcox [16], in most cases this can be done with a compile time data structure called here an *operand stack*. This data structure is a stack of data descriptors that simulates the run-time expression stack of an idealized machine executing postfix expressions.

By inspecting the top few data descriptors on this stack, the code generator for Euclid is able to reduce the source code shown on the left to the PDP-11 machine code idioms shown on the right:

$i := -i;$	NEG i
if $i >= 1 \ldots$	TST i
$i := i - 1;$	DEC i
$i := 2 * i;$	ASL i
$i := 0;$	CLR i

The operand stack allows a "window size" of two operators and three operands without a great deal of complexity. This window size is sufficient to match most machine idioms of typical architectures.

5. HIGH-LEVEL ADDRESSING

We will now show how the code optimization techniques that we have described are used in generating code to support displays and high-level addressing. First we show how data descriptors containing lexical bases are translated to data descriptors acceptable to machine instructions. Then we show how high-level operations, such as subscripting and field selection, are translated to machine instructions.

5.1 Resolving Lexical Bases and Forcing Addressability

The allocator we described produces data descriptors of the form $@^k Ln.d$ where $k = 1$ or 2 (also of the form Rn if the allocator is smart enough to put scalar locals in registers). Two steps are taken to change $@^k Ln.d$ to a form acceptable in machine instructions:

(1) *Resolve the lexical base.* We must remove Ln from the data descriptor, transforming the descriptor into an equivalent one whose base is a (possibly null) register.

(2) *Force addressability.* With L*n* resolved, data descriptors have the form @^{k}R*n.d.* As a result of generating code for operations such as addition, these data descriptors are manipulated; for indexed architectures, the indexed form @^{k}R*n.d.Im* may be created. Before emitting a particular instruction, we must force the data descriptor to have an addressing mode acceptable to the instruction. We call this "forcing the operand to be addressable."

5.2 Resolving Lexical Bases

We now show how to resolve a lexical base. The idea is to find the value of the data descriptor's lexical base and effectively insert this value into the descriptor's base. For example, in @L2.4 suppose that the descriptor associated with L2 is R5. Then @L2.4 is resolved to @R5.4.

If the target machine has enough registers, the compiler writer may choose to make each lexical base correspond to a fixed register; for example, L2 always corresponds to register R5. If this is the case, resolving a lexical base requires nothing more than a table look up of the corresponding register. But, in general, it is convenient to allow each lexical base to be represented by an arbitrary data descriptor. We will now consider this general case.

The data descriptors associated with lexical levels for a particular scope are kept in a data structure called a *compile-time display.*

Consider the example of Figure 2, in which we are compiling procedure Q's body. Lexical levels 1 (for P) and 2 (for Q) are active. The compile time display might be

L1: 342
L2: R5

Lexical level L1, for the main program, is based at absolute location 342. Variable *x* with data descriptor @L1.2 is resolved to @null.344 or simply @344. In this case, we cannot put the lexical value 342 into the base field because it is not a register, so instead we add it to *x*'s displacement.

To make the example more interesting, suppose procedure P3 is nested inside Q and procedure P4 is nested inside P3, and that compilation is presently inside P4. The compile-time display should have entries for four levels, and might be

L1(P): 342
L2(Q): @L3.-2
L3(P3): @L4.-2
L4(P4): R5

As is done in various Pascal compilers [17], a particular register R5 always addresses the local scope. Intermediate scopes are addressed by chaining down through "static links" from the local scope to the next surrounding scope and so on. The prologue of each procedure or function must establish these links and must set R5, while the epilogue must reset R5. The compiler must update its compile-time display when compilation crosses a scope boundary, so that this data structure always represents the corresponding run-time method of addressing.

We have shown negative displacements for levels L2 and L3. This would result in poor code for architectures such as the IBM 360 that allow only positive

```
procedure ResolveLexicalBase(var DD, resultDD: DataDescriptor);
  var saveIndir: integer;
  begin
    if DD.baseKind = lexicalBase then
      begin
        ResolveLexicalBase(DD.base↑, resultDD);
        saveIndir := DD.indir;
        DD.indir := 0;
        DD.baseKind := null;
        SuperAdd(DD, resultDD); {Add DD with null base, k = 0}
        DD.baseKind := lexicalBase; {Restore DD's base}
        DD.indir := saveIndir; {Restore DD's indirection k}
        resultDD.indir := saveIndir {Restore indirection}
      end
    else
      resultDD := DD
  end;
```

Fig. 5. Machine-independent recursive algorithm to resolve lexical bases.

displacements in instructions. But architectures such as the PDP-11 support signed displacements, and a negative displacement is as efficient as a positive one.

Lexical base L3 has descriptor @L4.-2, meaning that the base of the third scope is pointed to by the word located 2 bytes before where R5 points. That is, the static link from level 4 to level 3 is @R5.-2. Similarly, the static link from level 3 to 2 is @L3.-2.

More elaborate examples of chaining in the compile-time display occur in Euclid, where entries in the display locate module data and type data, as well as activation records [8].

Figure 5 gives a procedure to resolve lexical bases. It is recursive because the data descriptor associated with a lexical base may itself have a lexic base. For example, consider the resolution of @L2.10, where @L3.-2, is associated with L2, @L4.-2 with L3, and R5 with L4. The procedure is initially invoked with @L2.10, then recursively with @L3.-2, then with @L4.-2, and finally with R5.

The procedure is machine independent and relies on a call to the super-add procedure to generate code (if necessary) to add data descriptors DD and resultDD, with the sum represented by the returned value of resultDD. The call to super-add is avoided if DD's base is already a register or null base.

This recursive procedure is essentially a transliteration of the recursive definition of data descriptors into an executable program.

When there are repeated accesses to a particular lexical level, this causes repeated resolution of the associated lexical base. To avoid generating code for each resolution, the compile-time display entry for that base can be allocated a temporary register that has been set to directly locate the level's run-time data [16]. The use of a register optimizes the code for each access to that level.

One of the most interesting advantages of the techniques described in this section derives from the isolation of basic addressing considerations in the compile-time display. This isolation, or information hiding, is important because it separates concerns about addressing from concerns about generating code for

arithmetic. The result is that a code generator can conveniently and efficiently handle static languages (such as FORTRAN, which does not generally use a run-time display) and more dynamic languages (such as Pascal, which needs a run-time display). For FORTRAN, each compile-time display entry is set up to establish static bases of data areas, while for Pascal, the entries can specify run-time registers. The decision of how many registers to allocate to a run-time display is also isolated, and can be modified by simply changing the set-up of the compile-time display.

5.3 Forcing Addressability

Once the lexical base of a data descriptor has been resolved, the descriptor has a form directly usable by the code generator for generating arithmetic. But before a particular machine instruction can be emitted, its operands must be forced to have addressing modes acceptable to the target architecture.

For "orthogonal" architectures such as the PDP-11, the operands of most instructions can have the same addressing modes. As a result, "force addressable" is a single procedure useful to support most instructions. The IBM 360 is less orthogonal in that different IBM 360 modes are required depending on the instruction and the operand position. For an architecture like the 360, a set of procedures, such as ForceSAddressability (for S mode) and ForceXAddressability (for X mode), are needed.

These procedures are necessarily machine dependent, but their structure is not. Each does a case analysis to see if the particular data descriptor is acceptable as an addressing mode. If it is acceptable, no action is taken. If not, appropriate code must be generated to simulate the extra addressing power present in the data descriptor. For example, the descriptor @@R3 is not supported on the 360, so an explicit load, L R3,0(R3), could be emitted to reduce the data descriptor to @R3, which is acceptable as the S mode 0(R3).

5.4 Machine-Independent High-Level Addressing

Languages such as Pascal provide the user with high-level addressing operations, namely, subscripting, dotting (field selection), and pointing (using the up-arrow operator). The "get address" operation is implied when an actual parameter is passed to a var parameter. In this section we give a machine-independent mapping from these high-level operations to low-level machine-dependent operations.

We will assume that our code generator already supports the following low-level operations:

(1) Add via super-add.
(2) Subtract via super-subtract.
(3) Multiply via super-multiply.
(4) Increase indirection, i.e., increasing k in $@^k b.d.i.$
(5) Decrease indirection, i.e., decreasing k in $@^k b.d.i.$

We will give the algorithm to translate subscripting into machine code. For $V[t]$ we want to generate efficient code equivalent to

$$@(@^{-1}V + (t - \text{Lower}) * \text{Size})$$

where *V*, *t*, Lower, and Size are data descriptors representing the array, a temporary giving the subscript, the array's lower bound, and the element's size, respectively.

The following procedure accepts these four data descriptors and generates efficient code. This procedure returns the data descriptor for the array element in *t* and leaves the other descriptors unchanged.

```
procedure Subscript(var vector, t, lower, size: DataDescriptor);
  var negLower: DataDescriptor;
  begin
    DecreaseIndirection(vector);              {Get vector's address}
    CompileTimeNegate(lower, negLower);       {Negate lower}
    SuperAdd(negLower, t);                    {t := t − lower}
    SuperMultiply(size, t);                   {t := t * size}
    SuperAdd(vector, t);                      {t := t + addr(vector)}
    IncreaseIndirection(t);                   {t := @t}
    IncreaseIndirection(vector)               {Restore indirection}
  end;
```

Generating code for dotting is also quite simple, as the following procedure shows. The procedure is called to compile the field selection *r.f* by passing the data descriptors for *r* and *f* to parameters base and field, respectively. The base's data descriptor is left unchanged. As field is accepted, it should have a null base and zero indirection. As field is returned, it represents the entire reference *r.f*.

```
procedure Dot(var base, field: DataDescriptor);
  begin
    DecreaseIndirection(base);     {Get base's address}
    SuperAdd(base, field);         {Compute field's address}
    IncreaseIndirection(field);
    IncreaseIndirection(base)
  end;
```

To compile the Pascal reference $p\uparrow$, the code generator simply invokes IncreaseIndirection(DD), where DD is the data descriptor for *p*. Analogously, DecreaseIndirection(DD) is called when the address of data descriptor DD is required.

5.5 An Example of Addressing Code

We now show how the algorithms we have given produce good code for addressing. We take as an example the Pascal fragment:

```
var a:
    record
      b: array 1 . . 5 of
        record
          e: char;
          c: array 1 . . 3 of Boolean
        end;
      f: integer
    end;
. . .
a.b[i].c[j] := false
```

We show how code for the assignment statement is generated. This statement can be represented as

$$@(\text{Addr}a + Db + (\text{R}i - 1) * \text{Size}b + \text{D}c + (\text{R}j - 1)*\text{Size}c) := \text{false}$$

where

Addr$a$ is the address of a, which we will take to be absolute location 250
Ri and Rj are temporary registers holding the values of i and j
Size$b$ = 4 is the size of elements of array b
Size$c$ = 1 is the size of elements of array c
Db = 0 is the displacement of field b
Dc = 1 is the displacement of field c

Straightforward translation of this statement without optimization produces a great deal of machine code, because there are four additions, two subtractions, two multiplications, and one assignment. We will show how one addition, one multiplication, and one "clear" can be generated to support the statement on the PDP-11.

We begin by inserting numeric values for Addr$a$, Size$b$, Size$c$, Db, and Dc and show two successive applications of super-add:

$$@(250 + 0 + (\text{R}i - 1) * 4 + 1 + (\text{R}j - 1) * 1) := \text{false}$$
$$@(250 + \text{R}i - 1) * 4 + 1 + (\text{R}j - 1) * 1) := \text{false}$$
$$@(250 + \text{R}i.\text{-}1) * 4 + 1 + (\text{R}j - 1) * 1) := \text{false}$$

Further applications of super-add, subtract and multiply reduce this to

$$@((Ri.247) + (\text{R}j.\text{-}1)) := false$$

The resulting PDP-11 code is

```
MUL #4, Ri
ADD Rj, Ri
CLR 246(Ri)
```

This is much better code than would be emitted by a naive code generator.

6. CONCLUSIONS

This paper has introduced the data descriptor notation. This notation allows the compiler designer to describe data and addressing in a way that is independent of both the source language and the target machine. Data descriptors provide the basis for a machine-independent storage allocation algorithm. They support local optimization of common arithmetic operations, including addition, subscription, and multiplication. Algorithms are given for translating general data descriptors into existing hardware addressing modes, and for translating high-level addressing operations, such as subscripting, into machine language. Data descriptors have been used in a number of production compilers. They have been a key tool in the rapid creation of highly portable and efficient code generators.

ACKNOWLEDGMENT

The referees for this paper provided valuable comments relating this paper to previous research. Innumerable improvements of data descriptors have come from D. Wortman, D. Crowe, I. Griggs, J. Cordy, B. Spinney, and C. Lewis. C. McCrosky [12] used these techniques in moving Concurrent Euclid to the VAX. These techniques have been used by Steve Tjiang to move Concurrent Euclid and Turing to the 370; by Brian Thompson to generate MC6809 code for Concurrent Euclid; and by Mark Mendell to retarget to a number of machines including the inscrutable Intel 432 and the clumsy 8088/8086. J. Cordy's Ph.D. dissertation [2] extends data descriptors in developing a scheme for menu-driven code generators. I am grateful to D. Barnard for his constructive criticism of a draft of this paper.

REFERENCES

1. CARTER, L. R. An analysis of Pascal programs and several block optimizations. Ph.D. dissertation, Dept. of Computer Science, University of Colorado, Boulder, 1980.
2. CORDY, J. R. An orthogonal model for code generation. Ph.D. dissertation, Report CSRI-177, Computer Systems Research Institute, University of Toronto, Toronto, Feb. 1985.
3. CORDY, J. R., AND HOLT, R. C. Specification of Concurrent Euclid. In *Concurrent Euclid, the Unix System, and Tunis*, R. C. Holt, Ed. Addison-Wesley, Reading, Mass., 1983, 243–297.
4. CONWAY, R. W., AND WILCOX, T. R. Design and implementation of a diagnostic compiler for PL/I. *Commun. ACM 16*, 3 (Mar. 1973), 169–179.
5. HOLT, R. C. Data descriptors for use by the emitter. Euclid Working Paper 58, Toronto Euclid Workbook, Computer Systems Research Group, University of Toronto, Toronto, June 1978.
6. HOLT, R. C., AND CORDY, J. R. The Turing language report. Report CSRG-153, Computer Systems Research Group, University of Toronto, Toronto, Dec. 1983.
7. HOLT, R. C., AND HUME, J. N. P. *Introduction to Computer Science Using the Turing Programming Language*. Reston, Reston, Va., 1984.
8. HOLT, R. C., AND WORTMAN, D. B. A model for implementing Euclid modules and prototypes. *ACM Trans. Program. Lang. Syst. 4*, 4 (Oct. 1982).
9. HOLT, R. C., WORTMAN, D. B., AND CORDY, J. R. S/SL: syntax/semantic language. *ACM Trans. Program. Lang. Syst. 4*, 2 (April 1982).
10. LEWIS, C. Some early results and documentation for Concurrent Euclid. Master's thesis, Dept. of Computer Science, University of Toronto, Toronto, 1982.
11. LOWRY, E. AND MEDLOCK, C. Object code optimization. *Commun. ACM 12*, 1 (Jan. 1969), 13–22.
12. MCCROSKY, C. D. Porting a syntax/semantic language compiler for Concurrent Euclid. Master's thesis, Computer Science Dept., Queen's University, Kingston, Canada, 1981.
13. MCKEEMAN, W. M., HORNING, J. J., AND WORTMAN, D. B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, NJ, 1970.
14. SPINNEY, B. A technique for the construction of portable production quality code generators. Master's thesis, Dept. of Computer Science, University of Toronto, Toronto, 1981.
15. WAITE, W. M. AND GOOS, G. *Compiler Construction*. Springer-Verlag, New York, 1984.
16. WILCOX, T. R. Generating machine code for high-level programming languages. Ph.D. dissertation, Computer Science Dept., Cornell University, Ithaca, NY, 1971.
17. WIRTH, N. Design of a Pascal compiler. *Softw. Pract. Exper. 1*, 4 (Oct–Dec. 1971), 309–333.
18. WULF, W., JOHNSSON, R. K., WEINSTOCK, C. B., HOBBS, S. O., AND GESCHKE, C. M. *The Design of an Optimizing Compiler*. Elsevier North-Holland, New York, 1975.

Received May 1984; revised June 1986; accepted August 1986