SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(9), 835–870 (SEPTEMBER 1994)

Assignment as the Sole Means of Updating Objects

P. A. BUHR, DAVID TILL
Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada, N2L 3G1 (pabuhr@uwaterloo.ca)

AND

C. R. ZARNKE
Hayes Microcomputer Products, 295 Phillip Street, Waterloo, Ontario, N2L 3W8 (bzarn_l@hayes.com)

SUMMARY

It is possible, by appropriate programming language extensions, to use the assignment statement as the sole means of changing the value of a variable, thereby eliminating the need to modify routine arguments by output parameters. Several suggestions are made to enhance the syntax and semantics of routine definition and assignment to maintain notational convenience and efficient execution normally associated with output parameters. Finally, an almost complete implementation of the ideas is presented in C.

KEY WORDS Assignment Parameter passing Programming language

INTRODUCTION

In conventional imperative programming languages there are two ways in which a variable can be assigned: by the assignment statement and by being an output argument of a routine*(e.g., Pascal[1] VAR or Ada[†][2] OUT and IN OUT parameters). Unfortunately, the notation of a routine call usually does not indicate if an argument is changed. As a result, arguments can be changed unexpectedly, making comprehension, maintenance and debugging of a program difficult. A way of expressing that an argument is changed at the call site was discussed in the preliminary Ada reference manual[3] (Sections 5.3-5.4) but was not included in the revised version of Ada. C[4] forces the address of an argument to be passed to a routine if the argument is to be reassigned. This approach is represented syntactically at the call site by prefixing the argument by an '&', but there are situations where this is unnecessary (e.g., an array parameter). For most imperative programming languages, it is necessary to examine the routine definition to know if an argument is changed. We propose that there be only one mechanism to change the value of a variable and, given the choice between the two mechanisms just described, it must be by assignment. However, this must be able to be accomplished without sacrificing notational convenience or efficient execution normally associated with output parameters. Achieving these goals requires augmenting both routine definition and assignment.

Both routine definition and assignment, as defined in languages such as Pascal and Ada, are relatively pedestrian. As will be shown, their utility can be increased substantially, but

* In this paper, no differentiation is made between functions and procedures. All programs are called routines, some of these may return values.
† Ada is a Registered Trademark of the U.S. Department of Defense

CCC 0038–0644/94/090835–36
©1994 by John Wiley & Sons, Ltd.

Received 10 July 1993
Revised 17 June 1994

this has implications in several parts of the programming language. And if these implications are not addressed by appropriate extensions of the programming language, enhancing routine definition and assignment achieves only a small advantage.

This paper is divided into two parts. The first part discusses, in general, our approach to extending routine definition and assignment, as well as some additional facilities that we believe are important. This part should interest programming language designers as it discusses several basic language issues. The second part describes an almost complete implementation, in C, of the ideas presented in the first part. The implementation is achieved using a translator. No compiler support is used so that some trade-offs are necessary and not all aspects of the design are implemented. This part should interest programming language implementors as it discusses difficult implementation issues in extending C with more powerful routine definitions and assignment.

EFFECTS ON ROUTINES

Because we insist that all assignment to variables be explicitly shown through the language's assignment mechanism, conventional output parameters of a routine are disallowed. These are replaced by our own style of output parameters, assignment to which is explicitly stated. Consequently, traditional procedures, returning values through parameters, must be rewritten as value returning traditional functions. To allow traditional output parameters to be rewritten in the form of returned results, routine syntax and semantics must be extended.

First, routines must be able to return a value of any data type defined in the programming language, which includes instances of records, arrays, classes (as in Simula[5]), and possibly files and routines. Traditionally, these data types have been passed to a routine by address and modified directly through this address in the body of the routine. Since we want to eliminate this, a routine must be able to return these data types so that they can be assigned in the calling program. Thus, the programmer must now think in terms of returning results that are subsequently assigned to variables through assignment. This idea is similar to the fundamental method of work in functional programming, without having to contrive recursive solutions because of the lack of assignment, and is easily imported into imperative programming, as in Scheme[6] and ML[7].

Second, each output parameter of a traditional routine can be used to modify its corresponding argument; if a routine has several of them, multiple results are returned from that routine. The ability to return multiple results using output parameters is one of the important uses of a traditional procedure over a traditional function in conventional languages. To accommodate this, routines must be able to return multiple results as in CLU[8], Mesa[9] and ML. Since these multiple results must be assigned to an equal number of variables or used in a context that requires an equal number of values, it is necessary to make several augmentation to the language to effectively support this. Mimicking returning multiple results by packing them into a record or an array that is subsequently returned is not an acceptable method of doing this; hence, we believe that a programming language must have both facilities: returning complex data types and returning multiple results.

Routine syntax

The syntax we propose for routine definition expresses the fact that some parameters are changed. The syntax chosen is as follows:

$$\text{RTN}\quad \underset{\substack{\uparrow\\ \textit{return value}\\ \textit{(output parameter)}}}{x : \text{INT}} \quad\leftarrow\quad \overset{\substack{\textit{routine name}\\ \downarrow}}{f} \quad \underset{\substack{\uparrow\\ \textit{parameter(s)}\\ \textit{(input parameter)}}}{(\ \ldots\ )}$$

Here the return value, called an *output parameter*, is explicitly named and is specified using the assignment symbol, ←. This syntax is similar to that in APL[10].

A list of output parameters is allowed, to support returning multiple results, as in:

```
RTN  x, y : INT, z : REAL  ←  g( ... )
```

Here three results are returned, two are integer values and one is a real value. The location of the output parameters and the use of the assignment symbol indicates that the parameter is to be modified. It also models the usage of the routine at the call site; the above routines would be invoked in the following way:

```
y ← f( ... )
x, y, z ← g( ... )
```

As well, the output parameters are explicitly named, which is consistent with multiple input parameters, so that individual output parameters can be initialized. Explicitly named output parameters have other benefits. They make removal of temporaries normally associated with returning complex values straightforward by using the storage of the output argument directly (discussed in detail shortly). Furthermore, flow analysis can guarantee that all output parameters are initialized before returning. A separate RETURN statement that specifies a value to be returned, as in:

```
RETURN x
RETURN x, y, z
```

is rejected because it has the potential to list arguments in the wrong order for returning multiple values. Therefore, a simple RETURN with no arguments and running off the end of a routine are our choices.

Input parameters

While we want to prohibit the changing of arguments, the input parameters might still be changed. However, if assignment to input parameters is allowed, all arguments must necessarily be copied. This causes problems for certain kinds of objects, such as semaphores or files, that cannot be copied either because it is not sensible to do so or because no copy operation is defined.

If assignment to input parameters is disallowed, copying of arguments is at the discretion of the compiler. The compiler can copy or use a reference to the argument depending on which is more efficient in time or memory space and this would be mostly invisible to a user. This restriction is also consistent with the idea that arguments cannot be modified and would make parameters correspond more closely with the mathematical concept of function parameters, which never change value in the body of the function.

However, if the compiler passes an argument by reference, it is possible for a user to modify an argument by aliasing, for example:

```
VAR x : TYPE
RTN f( p : TYPE )
   x ← 1
   write( p )
END RTN
f(x)
```

This program generates different results if the compiler chooses to copy or use a reference to the argument of f. Euclid[11] showed that many aliasing situations can be controlled; Ada took the approach that it is the user's responsibility not to alias when it may violate the language's semantics. While the failure to guarantee that arguments cannot be changed is unfortunate, we believe it will occur infrequently, and does not negate the general benefits given above. Therefore, our design adopts read-only parameters, i.e. parameters are C style const.

It might be argued that our design may force extra local variables in which to copy the input parameters so that calculations can be performed. Introduction of new variables in this case increases complexity in the program. However, this situation is mitigated by the fact that the design has explicit output variables that can be modified. An input parameter can be copied into an output parameter and calculations can be performed directly on it there. Finally, we conjecture, from a simple analysis of our own and other programs, that most routines only read their parameters.

Even if direct assignment to a parameter is disallowed, it does not address the issue of changing values indirectly accessible through the parameter. We leave this to the programmer to specify in the type of the parameter using the CONST qualifier, as in:

```
RTN f( p : REF INT, q : REF CONST INT )
  *p ← 3            // valid
  *q ← 3            // invalid: value that q points at is constant
  p ← q             // invalid: input parameter cannot be changed
  q ← p             // invalid: input parameter cannot be changed
END RTN
```

Similarly, if the elements of an array are not defined to be constant or the parameter does not define them to be constant, they can be changed in the routine, but the entire array cannot be reassigned. If this scheme is not adopted, objects like a semaphore would be useless because a semaphore *must* be passed by reference and *must* be directly changed.

Output parameters

In the case of the output parameter, there are two reasonable interpretation for passing the output parameter from the routine to the output argument at the call site: write-only or read-write parameters (Ada style OUT or IN OUT, respectively). With write-only output parameters, the output parameter is treated as a local variable in the routine, and its value is assigned to the corresponding output argument. As in the case of input parameters, the compiler is free to choose the mechanism that is most appropriate for the type of the argument, that is, copy the value or pass the address of the output argument to the output parameter and modifying the argument directly. In the latter case, the output argument should be initialized to undefined, if possible, before usage of the output parameter since the output parameter is considered to be a local variable that is created without a value. As in the case of input parameters, aliasing could be a problem depending on the language.

The other alternative is a read-write output parameter. This means that the output parameter would be initialized to the value of the output argument (either by copying or by directly referencing it). However, in situations like the following:

```
y ← f(g(h(x)))
```

y could be uninitialized, and within the nested calls, it is necessary to create temporary variables with the same type as the output parameter to hold the results of the routine calls, as in:

```
t1 ← h(x)
t2 ← g(t1)
y ← f(t2)
```

In this situation, it is the temporary that is passed to routines g and h and the temporary is definitely uninitialized. Without a programming language facility to check for an undefined value, the read-write approach is too dangerous as users cannot rely on the output parameter to have a value. As well, this parameter passing mechanism would allow routines to be written in the following way:

```
RTN  s : [100] INT  ←  sort()
```

This version of sort gets its input from the output argument and modifies it directly. It would be called as follows:

```
y ← sort()
```

However, passing input into a routine through the output argument is arcane and does not follow intuitively from other notions in the programming language. For these reasons the read-write output parameters are rejected.

Based on this analysis, we adopt write-only output parameters. This decision follows conventional programming languages in requiring the allocation of the return value at the call site (output argument on the stack), where it is subsequently freed implicitly. This is in contrast to languages like Smalltalk[12] where it is the responsibility of a routine to create the object that is being returned. While this does allow an object to decide which of a group of compatible objects are returned, it makes storage management complex. For example, in Smalltalk, a simple integer or a multi-precision integer may be returned; these data values must be allocated dynamically, and hence, must be created on the heap, negating the efficiency of a program stack.

Optimizations

The routine extensions suggested above are, in general, no more expensive than traditional output parameters either in time or in space because output parameters are the same as Ada style OUT parameters. Therefore, it is easy to see how routines using traditional parameter passing modes can be re-written using the scheme presented here. For example:

```
PROCEDURE MatrixMultiply(a, b : IN Matrix; product : OUT Matrix)
```

becomes:

```
RTN  product : Matrix  ←  MatrixMultiply(a, b : Matrix)
```

A routine to sort an array of integers, written as:

```
PROCEDURE sort(x : IN ARRAY(1..100) OF INTEGER;
               s : OUT ARRAY(1..100) OF INTEGER);
```

becomes:

```
RTN  s : [100] INT  ←  sort(x : [100] INT)
```

and would be used in the following way:

```
s ← sort(y)
```

However, in the following situation:

```
s ← sort(s)
```

there is an aliasing of two names in the routine to a single data item in the caller's program. This presents the problem that during the sorting of the data values the input parameter values are changing as assignments are made to the output parameter. As stated, problems of this sort cannot be completely detected; nevertheless, in many cases the compiler can detect them and perform some reasonable action. One solution after detecting the aliasing is to make a temporary copy of the input parameter, as in:

```
s ← sort(s)    becomes    t ← s
                          s ← sort(t)
```

A novel alternative allows specifying that the input parameter is the same as the output parameter, as in:

```
RTN  x : [100] INT  ←  sort(x)
```

The routine writer is indicating that the input and output arguments must be the same. Here the name x appears as both an output and an input parameter to the routine; hence, such a parameter is called an *in-out parameter*. Since the type of x is defined by the output parameter, no type information is necessary when it appears as the input parameter. One consequence of this approach is that it is no longer sufficient to just specify parameter types in a routine prototype declaration; some parameter names must also be specified so both a programmer and the compiler know when an in-out parameter is being used.

The situation:

```
s ← sort(s)
```

is now allowed and it is obvious within routine sort that changes to the output parameter affect the input parameter because they are the same variable. This is identical to the traditional sort written with only a single IN OUT parameter, as in:

```
PROCEDURE sort(x : IN OUT ARRAY(1..100) OF INTEGER);
```

However, the situation:

```
s ← sort(y)
```

is now in conflict with the routine definition because the input and output arguments are not the same. Again, the compiler can detect this and make a copy of the input argument, as in:

```
s ← sort(y)   becomes   s ← y
                        s ← sort(s)
```

thus solving the conflict. Notice that it is unnecessary to create a temporary; instead it is only necessary to copy the input argument into the output argument and the output argument is passed as the input argument satisfying the routine requirement.

This facility can further eliminate temporaries in situations like nested routine calls, where the value being passed among the routines is the same type and size, such as several transformations to a matrix of values, for example:

```
y ← f(g(h(x)))   becomes   t1 ← h(x)
                           t2 ← g(t1)
                           y ← f(t2)
```

and two temporaries are created. For a compiler to optimize out the temporaries with traditional IN OUT parameters, it must perform flow analysis over the body of each routine to establish a direct relationship between the returned value and input value. If the body of the routine calls other routines whose bodies are not available, the analysis may fail. Furthermore, if the body of the routine itself is not available, the analysis cannot be done. With in-out parameters, the relationship between returned value and input value is explicitly stated in the interface, and therefore, the compiler is told this fact rather than having to infer it, plus the information is in the routine interface so it is always available to the compiler. For example, if the input and output parameters are the same for routines f, g and h, it would be possible for the compiler to trivially follow the chain of arguments, input and output parameters from x to y solely through routine interfaces. The variable y could then be used as the output data area (or temporary) for the calls of h and g, and f would use y as its output parameter to complete the expression, as in:

```
y ← f(g(h(x)))   becomes   y ← x
                           y ← h(y)   y is not copied because it is
                           y ← g(y)   being referenced by address
                           y ← f(y)
```

We suggest that both mechanisms for dealing with the aliasing problem be present in the programming language, because each method provides the programmer with a different mechanism for solving a problem. The programmer picks a routine form, based on the expected 'normal' case, and the compiler deals with the other cases. The latter solution, making the output and input parameter the same, is a novel way of providing additional information in the routine interface for optimization purposes, and allows the compiler to easily generate code for nested routine calls that is equivalent to the traditional IN OUT parameter case.

Conformant data parameters

Input parameters may be conformant arrays, i.e. the size of such input parameters is determined by the corresponding argument. However, in the case of conformant arrays in the output parameters there is no existing array from which to obtain the size, for example:

```
RTN  x : [] INT  ←  f( ... )
RTN  h( p : [] INT )
h( f( ... ) )
```

In the nested routine calls, the size of parameter p cannot be determined from the output parameter x. However, if the parameter, x, is an in-out parameter, as in:

```
RTN  x : [] INT  ←  f( ... , x, ... )
```

it is possible for the compiler to examine the argument that corresponds to the input parameter, x, to determine the size of the conformant parameter. For example, in:

```
VAR z : [100] INT
h( f( ... , z, ... ) )
```

the size of the argument, z, is used as the size of the conformant parameter (and possibly the size for a temporary variable to hold the result of f). The compiler starts at the innermost nesting level and propagates the actual array size for any conformant output parameter to outer nesting levels.

If the output parameter is a conformant array and not an input parameter, the size for the output parameter must come from the variable that receives the value of the output parameter, as in:

```
VAR   z : [100] INT
RTN  g(p : [100] INT)
RTN  x : [] INT  ←  f( ... )
g( f( ... ) )              // output parameter of f is the size of input parameter of g
z← f(...)                  // output parameter of f is the size of z
```

The variable to receive the result of the output parameter can be further away due to intervening calls to routines with input and output parameters that are the same, as in:

```
RTN  x : [] INT  ←  f(x)
RTN  x : [] INT  ←  g(x)
RTN  x : [] INT  ←  h( ... )
z ← f( g( h( ... ) ) )
```

The size of the output parameter is the size of z. As well, because of the routine definitions, the compiler does not need temporaries; instead the storage for z is used. Hence, the compiler starts at the innermost nesting level and instead of propagating the actual array size as in the previous case, it must look at the outer nesting levels for an actual array size to propagate back to an inner nesting level. Which direction to look is determined by whether the output parameter is also an input parameter.

Collectively, these extensions to routine syntax and semantics allow a traditional routine with IN OUT parameters to be rewritten as a routine returning results easily and without any loss, and possibly a slight gain, in efficiency. However, to benefit fully from these extensions requires modifications to other parts of the programming language so that they can be used to their fullest potential.

EFFECTS ON ASSIGNMENT

Traditional assignment is too limiting because it allows assignment of only one value to one variable or to part of one variable. This section describes some possible extensions.

Statement versus operator

Most conventional programming languages (Basic, Fortran, Cobol, Modula-2/3, Pascal, Ada) specify assignment as a statement, which has the general form:

```
variable ← expression
```

Some languages (Algol-60, Algol68[13], C, C++) treat assignment like a conventional arithmetic operator, thereby allowing it to produce a value. This form can be used in any programming language context where a value is required, for example:

```
VAR  r : REAL
VAR  i, j : INT
VAR  a : [10] INT
RTN f(p, q : INT)
i ← j ← 5
a[ i ← 5 ]
f( i ← i + 1, j ← j + 1 )
```

Here, the value produced by assignment is used as the operand of another assignment, as a subscript and as an argument value. Finally, Mesa allows assignment to appear either as a statement or as an expression operator; it will be shown that the difference is used to enforce restrictions on certain kinds of assignment.

The main issue with assignment as an operator is whether the programming practice that it encourages is desirable. (C is notorious for this style of programming.) We contend that it is not desirable because the assignment operator may be hidden within an expression, and this somewhat contradicts our objective of making assignment apparent. This style makes finding the places where a variable is modified difficult, and hence, is nearly as undesirable as modifying arguments of routines invoked in the middle of expressions. As well, there are some situations where the result may be indeterminate, as in:

```
( i ← j ) + ( j ← k )
a[ i ← 1 ] ← i + 1
f( i ← i + 1, i ← i + 1 )
```

Depending on the order that the compiler performs the assignments, the results will be different. For these reasons, we believe it is reasonable to restrict assignment to appear only in a separate statement and not allow it to be used as an operator. Such a restriction neither impedes the ability to write efficient programs nor prevents the compiler from generating efficient code.

Mass assignment

Some languages, like PL/I[14], allow several variables to be assigned a particular value, called *mass assignment*, for example:

$v_1, v_2, ..., v_n \leftarrow$ expression

where the v_i's are variables. Mass assignment is particularly useful for initializing variables, as in:

```
i, j, k ← 0
```

We define mass assignment to have *parallel semantics*, that is, the right-hand side expression is evaluated and assigned to each variable on the left-hand side as if all assignments occurred simultaneously. Operationally this means:

```
temp $\leftarrow$ expression
$v_1 \leftarrow$ temp
$v_2 \leftarrow$ temp
...
$v_n \leftarrow$ temp
```

rather than:

```
$v_n \leftarrow$ expression
$v_{n-1} \leftarrow v_n$
...
$v_1 \leftarrow v_2$
```

which avoids problems when the variables are of different type. For example, given r a real variable and i an integer variable then if:

r, i $\leftarrow$ 3.5   means   i $\leftarrow$ 3.5; r $\leftarrow$ i

r obtains the value of 3 instead of 3.5. This interpretation of the statement is too arcane. It is not normally what a programmer desires and it is difficult to locate this kind of mistake. Conversions from one type to another that may result in a loss of information should be done with an explicit cast or coercion facility.

Multiple assignment

Some languages allow a list of values in assignment, as in:

$v_1, v_2, ..., v_n \leftarrow e_1, e_2, ..., e_n$

where the v_i's are variables and the e_i's are expressions. We call this *multiple assignment* (also called *concurrent assignment*[15]). One such language is BCPL[16] and its meaning is:

```
$v_1 \leftarrow e_1$
$v_2 \leftarrow e_2$
...
$v_n \leftarrow e_n$
```

As for mass assignment, we believe this meaning for multiple assignment is arcane, as the following:

x, y $\leftarrow$ y, x

does not do anything sensible. In CLU, multiple assignment has parallel semantics, which means:

```
$t_1 \leftarrow e_1$
$t_2 \leftarrow e_2$
...
$t_n \leftarrow e_n$
$v_1 \leftarrow t_1$
$v_2 \leftarrow t_2$
...
$v_n \leftarrow t_n$
```

so that:

```
x, y ← y, x
```

interchanges the values of x and y. Normally interchanging two variables requires explicit creation of a temporary variable; in general, eliminating variables correspondingly reduces program complexity. Therefore, we adopt parallel semantics for multiple assignment.

However, in CLU, a variable may appear only once on the left-hand side of a multiple assignment statement. We could find no explanation for this restriction; presumably, the order in which the assignments are performed is conceptually simultaneous, so a statement such as:

```
x, x ← x, x+1
```

has no clear meaning because the ordering of the individual assignments by the compiler could be indeterminate. We, too, dislike statements with indeterminate semantics, because we believe programmers find them confusing, particularly at this fundamental level, even though they may afford the compiler an additional degree of freedom for code generation. A classical example appears in C, where f(i++, i++) has indeterminate semantics because the order of evaluation of arguments is not defined. Therefore, we define a left to right ordering in the evaluation of operands of an operator and in the execution of multiple assignment so the previous statement means:

```
t₁ ← x; t₂ ← x+1
x ← t₁; x ← t₂
```

and the value of x is incremented. Finally, performing assignments from left to right for multiple assignment is consistent with other forms of assignment that are discussed.

Only the combinations of assigning several values to the same number of variables or of assigning one value to several variables (mass assignment) are allowed, as the semantics of these are clear. Other combinations are not allowed as the meaning is unclear, for example:

```
a, b, c, d ← x, y          // what does this mean ?
a, b ← x, y, p, q          // what does this mean ?
```

Icon[17] defines a special exchange operation that can be used to swap two variables, for example, x :=: y exchanges the values of x and y. However, multiple assignment subsumes this case and is much more powerful. Mesa has a restricted form of multiple assignment *only* for record types, as in:

```
birthday : RECORD [
                  day   : [1..31],
                  month : [1..12],
                  year  : [1900..2050)
               ]
[dd,mm,yy] ← birthday;          -- extractor
birthday ← [dd,mm,yy];          -- constructor
```

Cascade assignment

Even with mass and multiple assignment, having one statement contain several assignments is still useful, as in:

```
v₁ ← v₂ ← ... ← vₙ ← expression
```

where the v_i's are variables. We call this *cascade assignment* (sometimes referred to as multiple assignment). As for mass and multiple assignment, problems with type conversions are mitigated by adopting parallel semantics so that cascade assignment means:

```
temp ← expression
v₁ ← temp
v₂ ← temp
...
vₙ ← temp
```

Cascade assignment is useful in situations where the expression on the right-hand side has a side effect. In this case, cascade assignment cannot be mimicked by multiple assignment, for example,

```
a ← b ← f(1)
```

cannot be simply transformed into:

```
a, b ← f(1), f(1)
```

because routine f is now evaluated twice. This is similar to the difference between the following two C statements:

```
a[random(10)] = a[random(10)] + 1;        /*  random evaluated twice */
a[random(10)] += 1;                       /*  random evaluated once */
```

Because cascade assignment may eliminate the explicit temporary variable needed if only multiple assignment were present in the language, we believe it is a useful construct.

Cascade assignment can be easily extended to work in combination with multiple assignment, as in:

```
a, b, c ← d, e, f ← f(1), g(2), h(2)
```

Like multiple assignment, only the combinations of assigning several values to the same number of variables or of assigning one value to several variables (mass assignment) are allowed.

Mesa has both cascade and (restricted) multiple assignment but they can only be combined in restricted forms. As mentioned, Mesa has an assignment statement as well as an assignment expression. The statement form of assignment is used to enforce the restriction that extractors can only appear as the left-hand side of an assignment statement; an extractor/constructor cannot appear as the left-hand operand of an assignment operator in an assignment expression, so the following is illegal:

```
birthday ← [dd,mm,yy] ← birthday;
```

Thus, Mesa does not provide a general mechanism for combining cascade and multiple assignment.

TUPLES

Further programming language extensions are suggested here to take advantage of the ordered list of heterogeneous items that have been introduced by routines returning multiple values and extended assignment. Both capabilities suggest that a list of variables or expressions be considered a separate construct in its own right. These lists of values are called *tuples*. While a tuple is a series of items of possibly different types, our tuple is not a record. It is never considered as a single structured set of data with separately accessible fields, as a record is.

SETL[18] provides a different form of tuple, which is an ordered sequence of arbitrary entities organized into a single entity using the following syntax:

```
[E₁, ..., Eₙ]
```

The entities E_i of a SETL tuple can be of any type and a tuple can be a component of another tuple, i.e., tuples can be nested. However, SETL essentially treats tuples as arrays of unbounded size, allowing values to be selected from and assigned to the tuple using subscripting, which is different from our approach. Finally, SETL allows the assignment:

```
[x₁, ... , xₖ] := t;
```

where x_1 through x_k are variables and t is a tuple, which is equivalent to:

```
x₁ := t(1);
...
xₖ := t(k);
```

and thus can be thought of as a multiple assignment.

ML provides a tuple context more like what we are suggesting. An ML tuple is a cartesian product of heterogeneous values. In essence, an ML tuple is a record, where the first field has the name ‘1’, the second field has the name ‘2’, etc., and the names must be consecutive integer values. Because ML tuples are records, they also have internal structure. ML functions allow only one argument and return value, but that argument or return value may be a tuple. ML also provides pattern matching to implicitly compose a list of values into a tuple, and to extract the fields of a tuple into individual values. Our tuples differ most significantly from those in ML by their lack of structure.

Existing tuple contexts

Depending on the particular programming language, there may be several syntactic contexts where tuples already exist. Two tuple contexts common to most imperative languages are the subscript list and argument list, for example:

```
VAR a : [10, 10, 10] REAL
RTN g(x, y, z : INT)
a[i, j, k] ← 3.0            // i, j, k form a tuple
g(4, 5, 6)                  // 4, 5, 6 form a tuple
```

Like ML, our definition of tuple allows a tuple to be used in a tuple context. For example, a routine returning multiple values, as in:

```
RTN  a, b, c : INT  ←  f( ... )
```

is actually returning a tuple and can be used in the following ways:

```
a[f( ... )] ← 3.0          // output parameters match with subscripts
g(f( ... ))                // output parameters match with input parameters
```

Multiple assignment affords another tuple context so that the following is possible:

```
x, y, z ← f(...)           // output parameters match with assignment tuple
```

In all these cases, the number of items in the tuple must match exactly with that required by the tuple context. An aesthetic problem is that the syntax does not show that the number of tuple items is equal, i.e., a routine call or subscript operation no longer indicates the number of input parameters or the dimension of the array. However, we believe that this capability provides a programming technique that is very expressive, overriding this small syntactic deficiency.

New tuple contexts

Tuples can be formed in ways other than simple lists of variables or expressions. One example is a short-hand form for referring to several fields of a record:

```
VAR r :  RECORD
             a, b, c, d, e : INT
         END RECORD
r.(a, b, c) ← 1, 2, 3   means   r.a, r.b, r.c ← 1, 2, 3
```

Here the record r is ‘distributed’ over the parenthesized list of field names. This form is particularly useful if r is subscripted or if several qualifiers are necessary, as in:

```
r[i, j].(a, d, c) ← 1, 2, 3    means   r[i, j].a, r[i, j].d, r[i, j].c ← 1, 2, 3
p.q.v.(b, e, d) ← 1, 2, 3   means   p.q.v.b, p.q.v.e, p.q.v.d ← 1, 2, 3
```

These forms do not obviate a statement like the Pascal WITH statement, but it does mitigate its use; the above two examples would be written in Pascal as:

```
WITH r[i, j] DO                  WITH p.q.v DO
   BEGIN                            BEGIN
   a := 1; d := 2; c := 3           b := 1; e := 2; d := 3
   END                              END
```

The former statements are more readable while still allowing the compiler to optimize the references as easily as it can in the WITH statement.

In general, this facility is used to ‘flatten’ a nested structure into a linear tuple. For example, given the following structure:

```
t : RECORD
  a : RECORD
    x : ...
    y : ...
  END RECORD
  b : ...
  c : RECORD
    z : ...
    w : ...
  END RECORD
END RECORD
```

this can be completely flattened by a tuple:

```
VAR v : t
v.( a.( x, y ), b, c.( z, w ) )
```

Here both variable and field names are used to specify the tuple. While it might be possible to construct an interpretation of a situation like:

```
v.(x, y)
```

it is not our intention to extend this idea further than flattening of records. For example, PL/I allows qualification to be omitted if it is unambiguous.

Tuple type

Each element of a tuple has a type. The type of a tuple, therefore, is the tuple of the types of the components of the tuple, for example, the type of tuple, 2, 4.3, is int, real. Two tuples are compatible if and only if they have the same number of components and corresponding components are type compatible. Because a tuple is flattened (defined shortly), its internal structure is unimportant in determining compatibility.

Tuple variables

The existence of tuples suggests a need for variables to store tuple values, for example:

```
VAR x : INT, REAL
```

More complex tuple variables can be created using other type constructors, such as pointers and arrays. Tuple variables can be used in assignments:

```
x ← 4, 3.1
i, r ← x
```

Note that tuple variables are not tuples. Tuple variables reference contiguous areas of storage, in which tuple values are stored; tuple variables and tuple values are runtime entities. Tuples are compile-time constructs, usually a list of expressions, whose values may not be stored contiguously.

Also, tuple variables are not record variables. First, the elements of the tuple variable are not individually accessible. Second, a tuple variable may be used in a tuple context, where a record variable cannot. This capability is possible because of several implicit coercion operations, discussed next. Finally, our tuples have a simple internal structure, a flat list, whereas records may have a complex internal structure.

Tuple coercions

There are four coercions that can sensibly be performed on tuples and tuple variables: closing, opening, flattening and structuring. In addition, the coercion of dereferencing can be performed on a tuple variable to yield its value(s), as for other variables.

Closing takes a tuple of values and converts it into a tuple value, which is a contiguous set of values. The following assignment illustrates closing:

```
VAR w : INT, INT, INT, INT
w ← 1, 2, 3, 4
```

First the right-hand tuple is closed into a tuple value and then the tuple value is assigned using a block copy.

Opening is basically the opposite of closing; a tuple value is converted into a tuple of values. The following assignment illustrates opening:

```
a, b, c, d ← w
```

w is implicitly opened to yield a tuple of four values, which are then assigned individually using multiple assignment.

Flattening coerces a nested tuple, i.e., a tuple with one or more components which are themselves tuples, into a flattened tuple, which is a tuple whose components are not tuples. For example, flattening the tuple 1, [2, 3], 4, where [ ] are used to enclose a nested tuple like in SETL, produces 1, 2, 3, 4. Flattening is also performed on tuple types. For example, the type int, [int, int], int can be coerced, using flattening, into the type int, int, int, int. The following assignment illustrates flattening:

```
VAR a, b, c, d : INT
a, b, c, d ← 1, [2, 3], 4
```

First the right-hand tuple is flattened and then the values are assigned individually using multiple assignment.

Neither SETL nor ML support flattening. In both languages, a tuple has internal structure (i.e., an array or record, respectively), and this structure must match exactly with the context in which the tuple is used. We believe structured tuples preclude many reasonable cases, both simple and complex. A simple case is:

```
VAR x : INT, REAL
x ← 1, 2.5
a, b, c, d ← 1, x, FALSE
```

In ML, the tuple x must be explicitly flattened:

```
val x = (1, 2.5);
val (a, b, c, d) = (1, #1x, #2x, 4);
```

A complex case is combining routines without having to restructure intermediate tuple values. For example, the following is not allowed in ML but allowed in our design:

```
RTN [x, y : int], z : int ← f( ... ) ...
RTN g( x, y, z : int ) ...
g( f( ... ) )            // implicit flattening coercion
```

In ML, the output parameter type of f: [int, int], int does not match the input parameter type of g: int, int, int. In our design, the call is valid because of the implicit flattening coercion. The lack of a flattening coercion may be a secondary issue in ML because the tuple is used for other purposes where flattening would cause problems; however, for our purposes, flattening is crucial to make tuples as general as possible.

Structuring is basically the opposite of flattening; a tuple is structured into a more complex nested tuple. For example, structuring the tuple 1, 2, 3, 4 into the tuple 1, [2, 3], 4 or the tuple type int, int, int, int into the tuple type int, [int, int], int. The following assignment illustrates all the coercion operations:

```
VAR w : INT, INT, INT, INT
VAR x : INT
w ← 1, 2, 3, 4
x ← 5
x, w ← w, x
```

w is opened, producing a tuple of four values; therefore, the right-hand tuple is now the tuple [1, 2, 3, 4], 5. This tuple is then flattened, yielding 1, 2, 3, 4, 5, which is structured into 1, [2, 3, 4, 5] to match the tuple type of the left-hand side. The tuple 2, 3, 4, 5 is then closed to create a tuple value. Finally, x is assigned 1 and w is assigned the tuple value using multiple assignment. A possible additional language extension is to use the structuring coercion for tuples to initialize a complex record with a tuple. Neither SETL nor ML support structuring, which results in similar restrictions to not having a flattening coercion. For example, the following is not allowed in ML but allowed in our design:

```
RTN [x, y : int], z : int ← f( ... ) ...
RTN g( x : int, [y, z : int] ) ...
g( f( ... ) )            // flattening and structuring coercions
```

Using tuple variables

We conjecture that tuple variables will not be used frequently; nevertheless, they have their uses. The following are two possible situations where tuple variables are useful. Tuple variables can be used to store argument lists for a routine call, as in:

```
VAR arg : INT, INT, INT, INT
arg ← 1, 2, a, b
WHILE ... DO
  IF x > 0 THEN
    f( arg )             // implicitly coerced open
  ELSE
    f( c, d, 3 ,4 )
  ENDIF
END WHILE
```

In this case, the argument list for the first call to f is formed in a tuple variable outside the loop and can be copied onto the calling stack with a block copy. Clearly, this optimization could be performed without tuple variables, but it is more difficult. The same optimization can be performed when accepting or returning multiple values, as in:

```
RTN ret : INT, INT, INT ← f( arg : INT, INT, INT )
```

If the tuple value passed as an argument is already closed or if the value returned does not need to be opened, an efficient block copy can be used.

ROUTINE PARAMETERS ON THE RIGHT SIDE OF ASSIGNMENT

A programming language extension is suggested to further increase the expressive power of the language, which follows naturally from the suggested extensions thus far. An output parameter of a routine is able to be associated with a variable on the left side of an assignment (as in x ← f(a)); this suggests the extension of allowing an input parameter of a routine to be associated with a variable on the right of an assignment[19]. Such a routine is defined in the following way:

```
RTN   f   ( ... )   ←    z : INT
              ↑              ↑
          input(s)       input(s)
         parameter      parameter
```

and is invoked by, f(...) ← 3, which is really ‘syntactic sugar’ for f′(... , 3), where f′ is the same as f with the additional input parameter:

```
RTN f′( ... , z : INT)
```

Hence z is just an input parameter and must obey all the rules of input parameters. Some languages (e.g. Algol68, C++, ML) allow a routine to appear on the left side of assignment, its meaning is quite different from that suggested here. Such a routine in Algol68 must produce a pointer (or REF) as its result that is used to receive the value of the right side of the assignment as opposed to the right side being an implicit parameter to the routine.

This extension can be used to increase the notational capability of the programming language by allowing nested routine calls to be written in the form of cascade assignment. For example, instead of the traditional form, x ← q(y, r(s(z))), these routines could be redefined:

```
RTN  out : ...  ←  s( ... )
RTN  out : ...  ←  r()  ← in : ...
RTN  out : ...  ←  q( ... )  ←  in : ...
```

so that they would be invoked in the following fashion:

```
x ← q(y) ← r() ← s(z)
```

The inputs to these routines can be divided into two classes: primary and secondary. The primary inputs are those values modified by the routine, while the secondary inputs provide information on how the primary inputs are modified. This form resembles the syntax used for pipes in the UNIX Shell[20] except that the data ‘flows’ towards the left rather than the right. However there is no implied concurrent execution; both these points are considered shortly. A more concrete example of this extension is the problem of determining if an array of words contains any misspelled words. A solution can be written in the following way using UNIX Shell commands[21]:

```
cat words | translit A-Z a-z | sort | unique | common -2 dict > mispelled
```

Using standard procedural notation this would be written:

```
mispelled ← common( -2, dict, unique( sort( translit( ′A-Z′, ′a-z′, words) ) ) )
```

and similarly written in the following way with the routine extension:

```
mispelled ← common( -2, dict) ← unique() ← sort() ← translit( ′A-Z′,′a-z′) ← words
```

Experience has shown that many people prefer the flattened pipe command syntax, rather than the nested routine call form; however, the nested form has historically been the only form available in imperative programming languages.

Finally, it is possible to have multiple input parameters by assignment. For example:

```
RTN r1, r2 : int ← foo( i1, i2 : int ) ← a1, a2 : int
```

returns two output parameters and accepts two conventional input parameters plus two input parameters by assignment. It is called in the following way:

```
x, y ← foo( 1, 2 ) ← x, 5
```

Note that the call must be consistent with the routine definition, so foo(1, 2, x, 5) is illegal.

Our approach is more consistent than that taken in the Shell or in C++. In the Shell, commands can allow input from both a filter or from an argument list, as in:

```
% cat  x | sort
% sort x
```

Because there is no formal definition for the parameters to a Shell command, there is no type-checking in commands. Hence, it is impossible to know which form is used, and the command must check if a parameter is supplied to it or not. A further problem occurs if both forms of input are specified, as in:

```
% cat x | sort y
```

This last form is a valid usage of sort as far as the Shell is concerned, but sort may not properly handle this situation, nor is it necessarily trivial to handle such a case.

In C++, overloading binary operators and other techniques are used to flatten expressions. For example, I/O operations are performed by overloading the binary shift operators to form an I/O pipeline, for example:

```
cin >> a >> b >> c;
cout << setw(5) << a << b << c;
```

However, defining a routine that fits in the pipeline with secondary parameters, e.g. setw(5), is difficult and non-obvious. Basically, a template is used to construct a closure around a routine with parameters to transform it into a manipulator routine with no parameters so it can be placed in the pipeline.

Input/output routines

The changes suggested have direct ramifications in the definition of input/output routines usually predefined in a programming language. Traditionally, input routines have used OUT parameters or a mechanism that looks like OUT parameters, as in:

```
READ(INPUT, A, B, C)      Pascal, output parameters
READ(5, *)  A, B, C       Fortran, a form of output parameters
```

These statements can be rewritten using output parameters, as in:

```
 a ← read(input)       a ← input.read
 b ← read(input)   or  b ← input.read
 c ← read(input)       c ← input.read
```

Similarly, output routines can be rewritten to use parameters on the right side of assignments:

```
write(output) ← a          output.write ← a
write(output) ← b    or    output.write ← b
write(output) ← c          output.write ← c
```

This latter change is not essential for output routines but is suggested to maintain the programming flavour from above.

This paper does not discuss how routines such as I/O routines would handle a variable number of parameters and/or polymorphic parameters. These facilities would allow the multiple statements above to be expressed as:

```
  a, b, c ← read(input)    or    a, b, c ← input.read
write(output) ← a, b, c    or    output.write ← a, b, c
```

A full discussion of these topics is beyond the scope of this paper. Some ideas on this topic have been presented[22].

DIRECTION OF ASSIGNMENT

It might be argued that the UNIX form for the spelling checker is more expressive because it reads left to right in the order that a programmer conceives of the operations. This is a valid point for interactive commands that are typed (left to right) as they are conceived, used and then discarded. And right to left assignment could be easily incorporated into a programming language, by having assignment assign to the variable on the right side, as in PROTEL[23] and BETA[24]:

```
a + b → c
sort(x) → x
```

Defining assignment to work this way does not affect any of the language extensions discussed so far. However, we believe that this syntax is less desirable when used with statements that will be read and modified at a later time because the most important component of assignment is the target, which is normally on the left, and hence, is able to be discerned easily when scanning the source code. If the targets are at the ends of ‘ragged right’ assignment statements, they are more difficult to find.

CONCURRENCY

Thus far, execution of assignment with routines has been assumed to be performed sequentially. This is not a property of assignment itself but of the definitions of the routines. If a language provided some mechanism to create tasks, such as a PROCESS construct:

```
PROCESS  x : [ ] INT  ←  foo( x : INT ) ...
```

which starts a new thread of control in a routine body (as in Mesa and Turing[25]) when it is called, concurrent execution of processes in an assignment statement could be made to operate very much like the Shell ‘|’.

One method for concurrent processes to communicate is through a pipe: an object that has routines read and write and an internal buffer, which when full causes suspension of the current writer or when empty causes suspension of the current reader. Figure 1 shows an example of a simple implementation of a pipe using a Simula-like CLASS. A more general definition

```
CLASS PipeOfChar
  ESCAPE EndOfData                // end of data exception
  VAR q : queue(CHAR, 10)         // bounded monitor queue of characters

  RTN ch :CHAR ← read()           // read characters from the queue
    ch ← q.front()                // return front element of queue
  END RTN

  RTN write( ch : CHAR )          // write characters into the queue
    q.back() ← ch                 //  add element to end of queue
  END RTN
  // initialization code for class
TERMINATION
  SIGNAL EndOfData                // termination code for class
END CLASS
```

Figure 1. Simple implementation of a pipe

for a pipe could be made by having a generic CLASS, which accepted the type of the queue elements as a parameter like the monitor queue above.

An example of the usage of this class is:

```
PROCESS  p : PipeOfChar  ←  Producer( ... )    ...
PROCESS  Consumer( ... )  ←  p : PipeOfChar    ...
VAR p : PipeOfChar

p ← Producer( ... )               // write into the pipe
...
Consumer( ... ) ← p               // read from the pipe
```

Producer begins writing into the pipe until it finishes or the pipe fills; Consumer subsequently removes the contents of the pipe, blocking if the pipe is empty. The more interesting situation is the following:

```
Consumer( ... ) ← Producer( ... )
```

where a single pipe is created (as a temporary) before starting Consumer and Producer, and Consumer uses this pipe directly as its input parameter. Here Producer and Consumer execute concurrently like Shell commands separated by a filter operator.

Figure 2 shows how input and output pipes can be used in a single process. This process would be used as follows:

```
Consumer( ... ) ← ProdCons( ... ) ← Producer( ... )
```

The exception EndOfData to signal end of input from the pipe requires handling. This is accomplished by the EXCEPTION control structure, which establishes the code to be executed for the escape EndOfData signalled in a lower-level block, in this case, the read routine of PipeIn. Eventually, when ProdCons is blocked on PipeIn.read(...) waiting for input, the process on the right of the assignment, Producer, will terminate execution. As part of its termination,

```
PROCESS  PipeOut : PipeOfChar  ←  ProdCons( ... )  ←  PipeIn : PipeOfChar
  VAR ch : CHAR
  ...
  EXCEPTION                              // catch signalled escape
    LOOP
      ch ← PipeIn.read()                 // read from pipe
      // process character ch, possibly writing more or fewer characters
      // into pipe PipeOut for each character from pipe PipeIn
      PipeOut.write(ch)                  // write into pipe
    END LOOP
  ESCAPE PipeIn.EndOfData
    // end of data processing for pipe PipeIn
  END EXCEPTION
  ...
END PROCESS
```

Figure 2. Input and output pipes

Producer ends its access to the pipe used to communicate with ProdCons; this is then signalled by the pipe and handled by ProdCons in the exception construct.

When this facility is used in conjunction with tuples, it is possible to construct multiple concurrent connections between two processes or between one process and several others, for example:

```
PROCESS  out₁, out₂ : PipeOfChar  ←  Producer()
PROCESS  Consumer()  ←  in₁, in₂ : PipeOfChar

Consumer() ← Producer()
```

Hence, there may exist multiple concurrent flows of information in an assignment statement.

C IMPLEMENTATION

A translator* for ANSI C, called K-W C† [26], was built to test the proposed ideas. C was chosen because of its popularity, however, we discovered that certain syntax quirks of C forced several significant extensions that would be unnecessary in other languages and also precluded some ideas completely. All K-W C extensions are directly applicable in C++.

Declarations

Early in the project, we discovered that it is impossible to extend existing C routine syntax to support multiple return values. This results from the fact that the routine name and parameters are embedded within the return type, mimicking the way that the return value is used at the routine's call site. For example, a routine returning a pointer to an array of integers, is defined and used in the following way:

* The term translator is used rather than preprocessor because the K-W C programs are partially parsed and symbol tables are constructed. A preprocessor normally does only string substitutions.

† 'K-W' stands for Kitchner-Waterloo, which is the twin city in which the University of Waterloo is located.

```
int (*y)[10];
int (*f( int (*x)[10] ))[10] { return x; };
...
(*f( y ))[3] += 1;          // definition mimics usage
```

While attempting to make the two contexts consistent was a laudable goal, it has not worked out in practice; C declaration syntax is notoriously confusing and error prone. Furthermore, this syntax cannot be extended with multiple return types because it is not possible to embed a single routine name within multiple return type specifications.

To allow the specification of multiple return types requires a completely new form of definition syntax for routines. The change to routine definition then affected other routine related declarations, such as routine prototype declarations and pointers to routines. Rather than have a declaration anomaly in K-W C just for routine specification, we attempted to extend all the C declaration contexts with the new form of declaration syntax. As it turned out, the syntax chosen for the extended routine syntax was amenable to all C declaration contexts, and is simpler to use than C declaration syntax. Furthermore, by making one small change to standard C declaration syntax, it was possible to allow our new declarations to appear with standard declarations; this backwards compatibility is useful because it allows K-W C code to coexist with the large body of existing C programs. In fact, the two styles of declaration may appear together in the same block, but cannot be mixed within a specific declaration. While extending the new declaration syntax to all declaration contexts is not germane to the thesis of this paper, it is simpler to explain the complete declaration change, rather than limit the discussion to routine contexts.

The one change made to standard C declaration syntax is to require a base type for all declarations, as is done in C++; C defaults to base type int if no type is specified, for example, the following are valid C declarations:

```
x;                      /*  int x */
*y;                     /*  int *y */
f( p1, p2 );            /*  int f( int p1, int p2 ); */
f( p1, p2 ) { }         /*  int f( int p1, int p2 ) { } */
```

Always specifying the base type is good programming practice. Furthermore, we believe that large amounts of existing C code would be unaffected by this change. This change is necessary to be able to distinguish between the new extended declarations and standard C declarations. The new declarations place all modifiers to the left of the base type, while standard C declarations place modifiers to the right of the base type. The only exception is bit field specification, which always appears to the right of the type modifier. If a base type is not made mandatory, it is impossible to tell if modifiers are for a new or a standard declaration.

In K-W C declarations, the character * is used to indicate a pointer, square brackets [ ] are used to represent an array, and parentheses ( ) are used to indicate a routine declaration, which is identical to C declarations. However, K-W C type declaration tokens are specified from left to right and the entire type specification is distributed across all variables in the declaration list. For instance, a variable x of type pointer to integer is defined in K-W C as follows:

K-W C	**C**
* int x;	int *x;

Other examples are:

```
 K-W C                 C
[20] int y;           int y[20];       /*  array of 20 integers */
* [20] float z;       float (*z)[20];  /*  pointer to array of 20 floats */
[20] * char w;        char *w[20];     /*  array of 20 pointers to char */
struct s {            struct s {
   int f0:3;             int f0:3;     /*  bit field syntax the same */
   * int f1;             int *f1;
   [10] * int f2;        int *f2[10]
};                    };
```

If a base type was not required, the declaration [10] *x; could be either a valid new declaration or an invalid old style declaration. As stated above, the two styles of declaration may appear together in the same block, but mixing is not recommended. The type modifiers extern and static are also supported.

Tuple types are discussed in detail in a later section.

Type operators

The new declaration syntax can be used in other contexts where types are required, such as casts and the pseudo-routine sizeof, for example:

```
 K-W C                       C
y = (* int)x;               y = (int *)x;
x = sizeof([10] * int);     x = sizeof(int *[10]);
```

Routine definition

The point of the new declaration syntax is to allow specifying routines that return multiple values, as in:

```
routine [int o1, int o2, char o3] foo(int i1, char i2, char i3) {
    routine body
}
```

which has three output and input parameters. First, the routine keyword is necessary to distinguish K-W C routine definition from routine prototype declarations.[*] Second, the type modifiers static and extern are possible before the routine keyword. Third, brackets, [ ], enclose the result type and each return value is named and that name is a local variable of the particular return type. The value of each local return variable is automatically returned at routine termination. Lastly, if there are no output parameters or input parameters, the brackets and parentheses must still be specified; in both cases the type is assumed to be void as opposed to the standard C default of int.

The routine foo could be called as follows:

```
[i, j, ch] = foo( 3, 'a', ch );
```

The list of return values from foo is treated as a tuple.

[*] We conjecture that it is possible to eliminate this keyword. Unfortunately, the way our grammar developed made subsequent attempts to eliminate it extremely difficult. We have left this issue for future work.

The syntax of the new routine prototype declaration follows directly from the new routine definition syntax; the type is the same, except the routine name and possibly parameter names are omitted, as in:

```
[int] () f;              /*  returning int with no parameters */
[* int](int) g;          /*  returning pointer to int with int parameter */
[](int,char) h;          /*  returning no result with int and char parameters */
[* int,int](int) k;      /*  returning pointer to int and int, with int parameter */
```

The decision to put the routine name at the end of the prototype instead of between the output and input parameters resulted from a desire to make a routine prototype declaration similar to a pointer to routine, as in:

```
[* int](int) *pg;  /*  pointer to routine returning pointer to int with int parameter */
```

This design decision is still under review.

In-out parameters require the name of an output parameter be specified but no type specified when it is used again in the input parameter list, as in:

```
[int, int y](y, int) bar;
routine [int x, int y] bar(y, int z) { ... }
```

Without requiring a base type in a declaration, it is impossible to tell an in-out parameter from a standard C declaration in the input parameter list.

Returning values

Because the value in the return variable is automatically returned when the routine terminates in K-W C, the return statement does not contain an expression, as in the following example:

```
routine [int x] bar() {
    ...
    x = 0;
    return;
}
```

When the return is encountered, the current value of x is returned to the caller. ‘Falling off the bottom’ of a routine is permitted, as in:

```
routine [int x] foobar() {
    x = 0;
}
```

Tuples

The general format of a tuple is the following:

[<*exprlist*>]

where <*exprlist*> is a list of one or more expressions separated by commas. The brackets, [ ], allow the parser to distinguish between tuples and expressions containing the C comma operator. The following are examples of tuples:

```
[x, y, z]
[2]
[v+w, x*y, 3.14159, foo()]
```

Tuples are permitted to contain sub-tuples; however, in virtually all contexts nested tuples are flattened, so a tuple, such as, [[14, 21], 9], which is a 2-element tuple whose first element is itself a tuple, is equivalent to the tuple, [14, 21, 9]. The only exception is when a tuple-structuring coercion is needed.

K-W C also considers the parameter list of a routine to form a tuple. Unfortunately, C's syntax for subscripts precluded treating them as tuples. The C subscript list has the form [i][j]... and not [i, j,...]. Therefore, there is no syntactic way for a routine returning multiple values to specify the different subscript values, for example, f[g()] always means a single subscript value because there is only one set of brackets. (Fixing this requires a major change to C because the syntactic form M[i,j,k] already has a particular meaning: i,j,k is a comma expression.)

K-W C supports tuple types and tuple variables, which can be used everywhere conventional types and variables can be used. The general format of a tuple type is:

[<*typelist*>]

where <*typelist*> is a list of one or more legal K-W C type specifications separated by commas, which may include other tuple type specifications. Examples of tuple types include:

```
[int, char]
[double, double, double]
[unsigned long]
[* [20] int, * * char, * [[int, int]] (int, int)]
```

Like tuples, tuple types may be nested, but also like tuples, nested tuple types are flattened because they are not a structuring mechanism like records, so the following tuple types are equivalent:

```
[int, int, int]
[[int, int], int]
[int, [int, int]]
```

Examples of tuple variables are:

```
[int, int] x;              /*  2 element tuple, each element of type int */
* [char, char] y;          /*  pointer to a 2 element tuple */
[[int, int]] ([int, int]) z;
```

The last example declares an external routine that expects a 2 element tuple as an input parameter and returns a 2 element tuple as its result.

Note, tuples that appear as parameter lists may have their square brackets omitted for convenience. Therefore, the following routine invocations are equivalent:

```
f( [1, x+2, fred()] );
f( 1,  x+2,  fred() );
```

Also, a tuple or a tuple variable may be used to supply all or part of a parameter list for a routine expecting multiple input parameters or for a routine expecting a tuple as an input parameter. For instance, the following are all legal:

```
[int, int] w1;
[int, int, int] w2;

[void] (int, int, int) foo;          /*  three input parameters of type int */
[void] ([int, int, int]) bar;        /*  3 element tuple as input */

foo( [ 1, 2, 3 ] );
foo( w1, 3 );
foo( 1, w1 );
foo( w2 );
bar( [ 1, 2, 3 ] );
bar( w1, 3 );
bar( 1, w1 );
bar( w2 );
```

Note that a tuple can contain a C comma operator, provided that the expression containing the comma operator is enclosed in parentheses. For instance, the following tuples are equivalent:

```
[1, 3, 5]
[1, (2, 3), 5]
```

The second element of the second tuple is the expression (2, 3), which yields the result 3.

Mass assignment

Mass assignment has the following general form in K-W C:

[<*lvalue*>, ..., <*lvalue*>] = <*expr*>;

The left-hand side is a tuple of <*lvalue*>s, which is a list of expressions each yielding an address, i.e., any data object that can appear on the left-hand side of a conventional assignment statement. <*expr*> is any standard arithmetic expression. Clearly, the types of the entities being assigned must be type compatible with the value of the expression.

Mass assignment has parallel semantics, for example, the statement:

```
[i, y[i], z] = a + b;
```

is conceptually equivalent to:

```
t   = a + b;
a1  = &i;
a2  = &y[i];
a3  = &z;
*a1 = t;
*a2 = t;
*a3 = t;
```

The temporary t is necessary to store the value of the expression to mitigate conversion problems. The temporaries for the addresses are needed so that locations on the left-hand side do not change as the values are assigned. In this case, y[i] uses the previous value of i and not the new value set at the beginning of the mass assignment. In general, a good optimizing compiler could remove most of the address temporaries by determining that individual assignments do not interfere.

Multiple assignment

Multiple assignment has the following general form in K-W C:

[*<lvalue>*, ..., *<lvalue>*] = [*<expr>*, ..., *<expr>*];

The left-hand side is a tuple of *<lvalue>*s, and the right-hand side is a tuple of *<expr>*s. Each *<expr>* appearing on the right-hand side of a multiple assignment statement is assigned to the corresponding *<lvalue>* on the left-hand side of the statement using the same parallel semantics as for mass assignment. K-W C's multiple assignment is more powerful than the CLU equivalent; in CLU, only variables can appear on the left-hand side of a multiple assignment statement.

An example of multiple assignment is:

```
[i, y[i], z] = [1, i, a + b];
```

Here, the values 1, i and a + b are assigned to the variables i, y[i] and z, respectively. Note that the parallel semantics of multiple assignment ensure that:

```
[x, y] = [y, x];
```

correctly swaps the values stored in x and y.

Finally, translator checks ensure that the number of entities in both tuples are equal. The following are flagged as errors:

```
[a, b, c] = [1, 2, 3, 4];
[a, b, c] = [1, 2];
```

The approach of issuing a warning and truncating and/or zero-filling the operands on the right-hand side of the assignment was rejected, since statements such as these are most likely the result of mistyping.

As for all tuple contexts in C, side effects should not be used because C does not define an ordering for the evaluation of the elements of a tuple; both these examples produce indeterminate results:

```
f(x++, x++);                /*  C routine call with side effects in arguments */
[v1, v2] = [x++, x++];      /*  side effects in RHS of multiple assignment */
```

Cascade assignment

Cascade assignment has the following general form in K-W C:

<tuple> = *<tuple>* = ... = *<tuple>*;

and it has parallel semantics as for mass and multiple assignment. Some examples of cascade assignment are:

```
x1 = y1 = x2 = y2 = 0;
[x1, y1] = [x2, y2] = [x3, y3];
[x1, y1] = [x2, y2] = 0;
[x1, y1] = z = 0;
```

As in standard C, the rightmost assignment is performed first, i.e., assignment parses right to left.

Record field tuples

Tuples may be used to select multiple fields of a record. Its general format is:

<expr> . [*<fieldlist>*]

<expr> is any expression yielding a value of type struct. Each element of *<fieldlist>* is an element of the struct specified by *<expr>*. A record field tuple may be used anywhere a tuple can be used. An example of the use of a record field tuple is the following:

```
struct s {
    int f1, f2;
    char f3;
    double f4;
};
struct s v;

foo( v.[f3, f1, f2] );          /*  equivalent to foo( v.f3, v.f1, v.f2 ) */
```

If a field of a struct is itself another struct, multiple fields of this subrecord can be specified using a nested record field tuple, as in the following example:

```
struct inner {
    int f2, f3;
};
struct outer {
    int f1;
    struct inner i;
    double f4;
};
struct outer v;

v.[f1, i.[f2, f3], f4] = [11, 12, 13, 3.14159];
```

Passing input parameters by assignment

K-W C allows input parameters to be passed in two ways: either in parentheses (the traditional way) or on the right-hand side of an assignment statement. It is possible to define routines that are called in the following way:

```
x = foo( 1 ) = [2, 3];
```

Here, the tuple [2, 3] is passed as an input parameter to foo. The following is a sample definition of a routine that accepts input parameters by assignment.

```
routine [int ret] foo( int parm1 ) = [int parm2, char parm3] {
    ...
}
```

Here, parm1 is a conventional input parameter, and parm2 and parm3 are input parameters passed by assignment.

Note that the two kinds of input parameters are not interchangeable. If the above routine foo is defined as shown, the statement:

```
foo( 1, 2, 3 );
```

is illegal, since foo is expecting only one conventional input parameter.

External declarations can specify input parameters passed both by assignment and in parentheses:

```
extern [int] (int) = [int, char] foo;
```

This declaration gives the routine prototype for foo. Routine types that specify input parameters passed by assignment can be used in other declarations as well, for example:

```
extern * [int] (int) = [int, char] fooptr;
```

K-W C TRANSLATION

The K-W C translator converts K-W C constructs into ANSI C, which can be subsequently compiled and executed. The purpose of the translator was to provide proof-of-concept for the ideas presented in the first half of the paper; while the translator is not complete, it has been used by students to test the ideas presented in the paper. Figure 3 shows a simple K-W C program that illustrates most of the K-W C features. The output from the K-W C translator for this program is presented to show the implementation techniques for the different features. Some editing of the translator output has been performed for display and readability purposes, and comments have been added.

Figure 4 shows the additional declarations added to the K-W C program. Each tuple and certain tuple contexts are transformed into a C structure. In the example program there are three structures generated: the implicit tuple returned from routine foo, the explicit tuple (tuple variable r) returned from routine bar, and the local tuple variable z in the body of routine bar. Routines foo and bar return instances of the first two structures, respectively.

Figure 5 shows the C code generated for routine foo. The output parameters appear as local variables in the body of the routine and are subsequently packed into a structure before returning. A compiler for K-W C would optimize out this step by using the storage for the routine result at the call site. The translator was not up to the task of locating the calls, and transforming the expression containing them so that the address of the call site result variable was passed as the first parameter to the routine.

The code for the multiple assignments show the three basic steps: calculating the addresses of the left-hand operands, copying the right-hand expressions, and assigning the right-hand values to the left-hand locations. The implicit return at the end of the routine copies the output parameters into the return structure, and that is copied back to the call site.

Figure 6 shows the C code generated for routine bar. Notice that the routine parameter that appears on the right-hand side of assignment has been made into a normal input parameter. Also, there are several contexts where tuples are flatten. In the first multiple assignment, the tuple variable z is flattened on the left-hand side, and the tuple returned from the call to foo is flattened on the right-hand side. In the second multiple assignment, the tuple output parameter r is flattened on the left-hand side, and the tuple variable z is flattened on the right-hand side.

Figure 7 shows the C code generated for routine main. The main routine illustrates the use of record field tuples. The implementation is straightforward, distributing the structure name across the field names in the tuple list. Also, in the call to routine bar, the input argument specified on the right-hand side of the assignment has been made into a normal argument.

Many of the features not supported in the translator relate to call site optimizations and block transfers. K-W C features requiring complex analysis at the call site and creation of call site

```
[int rc]( * char fmt, ... ) printf;

routine [short x, unsigned y] foo( int w ) {
    [y, x] = [x, y] = [w, 23];              /*  cascade, multiple assignment */
}

routine [[int, char, long, int] r] bar() = [short p] {
    short x;
    unsigned int y;
    [int, int] z;                           /*  tuple declaration */

    [x, y, z] = [p, foo( 17 ), 3];          /*  complex tuple coercions */
    r = [x, y, z];
}

routine [int rc] main( int argc, ** char argv ) {
    struct {
            int f1, f2, f3, f4;
    } s;

    s.[f1, f2, f3, f4] = bar() = 4;         /*  input argument by assignment */
    printf( "expecting 3, 17, 23, 4; got %d, %d, %d, %d\n", s.[f4, f3, f2, f1] );
    rc = 0;
}
```

Figure 3. Sample K-W C program

temporaries are not implemented, along with some tuple coercions. At this level, the translator is fighting with the compiler to do things very differently from the base language's semantics. Since most of these optimizations would be straightforward if the K-W C constructs were implemented directly in the compiler, we felt it was not worth expending a large effort to achieve them outside of the compiler. The following examples illustrate some of these cases.

The present implementation of K-W C does not support nested routine calls when the inner routine returns multiple values; i.e., statements such as f(g()) are not supported, where routine g returns multiple values or a tuple value. To implement this, the translator would have to pull the expression apart, store the tuple returned from g, and pull the values apart again to pass them to f. A compiler might notice that the layout of the temporary tuple returned from g is the same as the parameter list of a routine and eliminate the temporary by building the result from g on the stack and calling f directly because the result forms the argument for the call.

In-out parameters are not implemented for similar reasons: they would require complex decomposition of call sites in expressions. While the K-W C translator has the symbol table information about in-out parameters, applying that knowledge at call sites would have been extremely difficult. A compiler would simply perform in-out inferencing at each call site and remove unnecessary temporaries accordingly as part of standard code generation.

Finally, the K-W C translator does not perform tuple-close coercions to achieve efficient block transfers; instead, individual field transfers are performed. For example, in routine bar,

```
struct _i1 {                          /*  tuple returned from foo */
   short _s1;
   unsigned int _s2;
};
struct _i2 {                          /*  tuple returned from bar */
   int _s1;
   char _s2;
   long _s3;
   int _s4;
};
struct _i3 {                          /*  tuple for bar::z */
   int _s1;
   int _s2;
};
int printf(char *fmt, ...);           /*  prototypes for all routines */
struct _i1 foo(int w);
struct _i2 bar(short p);
int main(int argc, char **argv);
```

Figure 4. Additional declarations

```
struct _i1 foo(int w) {
   short x;
   unsigned int y;
   {                                  /*  [y, x] = [x, y] = [w, 23]; */
      short *_L1; unsigned int *_L2;
      int _R1; int _R2;
      _L1 = &x; _L2 = &y;             /*  addresses for LHS expressions */
      _R1 = w; _R2 = 23;              /*  copy RHS expressions */
      *_L1 = _R1; *_L2 = _R2;         /*  assign RHS to LHS */
   }
   {
      unsigned int *_L1; short *_L2;
      short _R1; unsigned int _R2;
      _L1 = &y; _L2 = &x;             /*  addresses for LHS expressions */
      _R1 = x; _R2 = y;               /*  copy RHS expressions */
      *_L1 = _R1; *_L2 = _R2;         /*  assign RHS to LHS */
   }
   {                                  /*  implicit return */
      struct _i1 _S;
      _S._s1 = x; _S._s2 = y;         /*  return output parameters */
      return( _S );
   }
}
```

Figure 5. Routine foo

```
struct _i2 bar(short p) {
    short x;
    unsigned int y;
    struct _i3 z;
    struct _i2 r;
    {                                       /*  [x, y, z] = [p, foo( 17 ), 3]; */
        short *_L1; unsigned int *_L2; int *_L3; int *_L4;
        int _R1; short _R2; unsigned int _R3; int _R4;
        _L1 = &x; _L2 = &y;                 /*  addresses for LHS expressions */
        {
            struct _i3 *_S;
            _S = &(z);                      /*  flatten tuple LHS */
            _L3 = &(_S->_s1); _L4 = &(_S->_s2);
        }
        _R1 = p;                            /*  copy RHS expressions */
        {
            struct _i1 _S;                  /*  flatten tuple RHS */
            _S = foo( 17 );
            _R2 = _S._s1; _R3 = _S._s2;
        }
        _R4 = 3;
                                            /*  assign RHS to LHS */
        *_L1 = _R1; *_L2 = _R2; *_L3 = _R3; *_L4 = _R4;
    }
    {                                       /*  r = [x, y, z]; */
        int *_L1; char *_L2; long *_L3; int *_L4;
        short _R1; unsigned int _R2; int _R3; int _R4;
        {                                   /*  addresses for LHS expressions */
            struct _i2 *_S;
            _S = &(r);                      /*  flatten tuple LHS */
            _L1 = &(_S->_s1); _L2 = &(_S->_s2);
            _L3 = &(_S->_s3); _L4 = &(_S->_s4);
        }
        _R1 = x; _R2 = y;                   /*  copy RHS expressions */
        {
            struct _i3 _S;
            _S = z;                         /*  flatten tuple RHS */
            _R3 = _S._s1; _R4 = _S._s2;
        }
                                            /*  assign RHS to LHS */
        *_L1 = _R1; *_L2 = _R2; *_L3 = _R3; *_L4 = _R4;
    }
    return( r );
}
```

Figure 6. Routine bar

```
int main(int argc, char **argv) {
    struct _t5 {
        int f1; int f2; int f3; int f4;
    };
    struct _t5 s;
    int rc;
    {                                       /*  s.[f1, f2, f3, f4] = bar(); */
        int *_L1; int *_L2; int *_L3; int *_L4;
        int _R1; char _R2; long _R3; int _R4;
                                            /*  addresses for LHS expressions */
        _L1 = &s.f1; _L2 = &s.f2; _L3 = &s.f3; _L4 = &s.f4;
        {
            struct _i2 _S;
            _S = bar((short)4);             /*  flatten tuple RHS */
            _R1 = _S._s1; _R2 = _S._s2;     /*  record field tuples */
            _R3 = _S._s3; _R4 = _S._s4;
        }
                                            /*  assign RHS to LHS */
        *_L1 = _R1; *_L2 = _R2; *_L3 = _R3; *_L4 = _R4;
    }
    printf( (( char * )"expecting 3, 17, 23, 4; got %d, %d, %d, %d\n"),
            s.f4, s.f3, s.f2, s.f1 );       /*  record field tuples */
    rc = 0;
    return( rc );
}
```

Figure 7. Routine main

the statement r = [x, y, z]; could have been implemented by copying the values of the right-hand side into a temporary of type struct _i2, which is subsequently assigned directly to r. A further optimization would be to remove the temporary and assign the individual values from the right-hand side directly to r. We felt that achieving these additional optimizations was unnecessary for demonstrating proof-of-concept for the basic ideas.

CONCLUSION

By extending the method for returning values from a routine, all variable modification can be accomplished using the assignment statement. As a result, all situations where a variable is modified are explicitly denoted by $\leftarrow$. This simplification is significant as it makes programs easier to read and modify, and simplifies teaching routines to beginning programmers because of the single consistent method of changing the value of a variable. To a large extent, this consistency can be attained without increasing execution time or using more memory.

A number of extensions are suggested by this methodology; some of these extensions have more general applications. Mass, multiple, and cascade assignments provide a convenient way of expressing what may take several statements in conventional programming languages. Tuples are a generalization of the lists of values introduced by multiple assignment and are important to make maximum utilization of routines that return multiple values. Treating some existing contexts as tuples, such as subscript and argument lists, makes tuples an orthogonal facility in the programming language. Implicit flattening and structuring coercions for our tuples enhances tuple assignment and functional composition. In ML, a programmer must explicitly decompose, recombine, and possibly create additional temporaries to achieve the same results. Allowing routines on the left side of assignment provides a notationally convenient way for nesting routine calls. When processes are introduced, concurrent execution using a communication mechanism like a pipe is possible, hence mimicking the filter operator in the UNIX Shell. This allows the convenience of combining programs together afforded in UNIX, which leads to convenient reusing of existing software.

In K-W C, we achieved most of the notational convenience from our initial design but not the performance benefits. The performance benefits would require changes in the compiler both in the type system and code generation. None of these changes are beyond current compiler technology but were beyond the scope of this project. Finally, the concurrency capabilities were not examined. Nevertheless, K-W C shows that the design is viable and can be incorporated into traditional programming languages even in a syntactically hostile language like C.

ACKNOWLEDGEMENTS

We would like to thank Gord Cormack and Glen Ditchfield for taking the time to read and comment on the ideas presented here. Also, referee A's comments resulted in a much improved paper.

REFERENCES

1. Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, Springer–Verlag, third edition, 1985. Revised by Andrew B. Mickel and James F. Miner, ISO Pascal Standard.
2. United States Department of Defense, *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag.
3. Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, Henry F. Ledgard, Jean-Claude Heliard, Jean-Raymond Abrial, John G. P. Barnes, and Olivier Roubine, 'Preliminary Ada reference manual', *SIGPLAN Notices*, **14**, (6), (June 1979). Part A.
4. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall Software Series, Prentice Hall, second edition, 1988.
5. O-J Dahl, B. Myhrhaug, and K. Nygaard, *Simula67 Common Base Language*, Norwegian Computing Center, Oslo Norway, October 1970.
6. Jonathan Rees and William Clinger, 'Revised^3 report on the algorithmic language Scheme', *SIGPLAN Notices*, **21**, (12), 37–79, (December 1986).
7. Robin Milner, 'A theory of type polymorphism in programming', *Journal of Computer and System Sciences*, **17**, 348–375, (1978).
8. Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder, *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*, Springer-Verlag, 1981.
9. James G. Mitchell, William Maybury, and Richard Sweet, 'Mesa language manual', Technical Report CSL–79–3, Xerox Palo Alto Research Center, (April 1979).
10. K. E. Iverson, *A Programming Language*, Wiley, New York, 1962.
11. B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek, 'Report on the programming language euclid', *SIGPLAN Notices*, **12**, (2), 1–79, (February 1977).
12. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
13. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisher, 'Revised report on the algorithmic language ALGOL 68', *SIGPLAN Notices*, **12**, (5), 1–70, (May 1977).

14. International Business Machines, *OS and DOS PL/I Reference Manual*, first edition, September 1981. Manual GC26-3977-0.
15. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.
16. M. Richards and C. Whitby-Strevens, *BCPL – The Language and Its Compiler*, Cambridge University Press, Cambridge, 1979.
17. Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*, Prentice-Hall, 1983.
18. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets: An Introduction to SETL*, Springer-Verlag, 1986.
19. A. J. W. Mayer, 'Value receiving procedures', *SIGPLAN Notices*, **16**, (11), 30–34, (November 1981).
20. S. R. Bourne, 'The unix shell', *The Bell System Technical Journal*, **57, part 2**, 1971–1990, (July–August 1978).
21. J. Bentley, 'A spelling checker', *Communications of the ACM*, **28**, (5), 456–462, (May 1985).
22. G. Ford and B. Hansche, 'Optional, repeatable, and varying type parameters', *SIGPLAN Notices*, **17**, (2), 41–48, (February 1982).
23. D. G. Foxall, M. L. Joliat, R. F. Kamel, and J. J. Miceli, 'PROTEL: A high level language for telephony', *Proceedings 3rd International Computer Software and Applications Conference*, November 1979, pp. 193–197.
24. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard, *Object-oriented Programming in the BETA*, Addison-Wesley, 1993.
25. R. C. Holt, *Turing Reference Manual*, Holt Software Associates Inc., third edition, 1992.
26. David W. Till, *Tuples In Imperative Programming Languages*, Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1989.