Technical Report CS74017-R

A NOTE ON LEDGARD'S MINILANGUAGE 2

AND

A PROPOSAL FOR AN ALTERNATIVE

John A. N. Lee*

October 1974

*Professor of Computer Science

Virginia Polytechnic Institute and State University

Blacksburg, Virginia

Abstract:

This note reviews and discusses the concepts of assignment statements which occur within programming languages and as exemplified by Minilanguage 2 by Ledgard [3]. Observing that the description of assignment by Ledgard depends on the existence of a nominative addressing scheme, an alternative vignette of language is proposed which is based on type directed addressing systems.

This report reflects research which was supported in part by the National Science Foundation, project number 74-294-03.

Key Words: Programming Language, Minilanguages, language design, compiler design, formal definition, semantics, assignment, addressing systems, nominative addressing, type directed addressing.

CR categories: 1.3, 1.52, 4.10, 4.22, 4.9

INTRODUCTION

With two exceptions, the description of 10 minilanguages by Ledgard [3] was achieved without reference to any particular set of machine dependent concepts or implementation schema. This is notable when viewed in the light of how languages have been influenced by architecture and conversely how languages can force applications implementations to be severely influenced by their underlying structure. The two exceptions are minilanguage 8 (Data Structures) and minilanguage 2 (Generalized Assignment). In the former Ledgard prescribed an information structure model which unfortunately severely limits the wider conceptual aspects of the language. That is, to assume that the components of any data structure are ordered leads to a lack of equivalence classes which would be highly desirable. The topic itself is worthy of another note.

Minilanguage 2 assumes (though does not explicitly require) a nominative [1] addressing structure in the underlying machine. That is, in the addressing cycle the attributes of the logically sequential data element are specified in the logical predecessor rather than being directly associated with the element itself. Thus indirect addressing is typically accomplished by the specification that the *next* logical

element contains an address which is to be used to locate the next data element. Even compound indirection, such as typified by the ML2 statement:

```
↓↓↓A:=↓↓B,
```

is nominative in implementation. If the following environment exists:

identifier	associated value
A	B
B	C
C	D
D	E

then the above ML2 statement evaluates to the equivalent assignment statement:

```
D:='E'
```

Even permitting the use of indirection over literals such that

```
↓'X' = X,
```

the nominative system of addressing is consistent.

Alternatively, we must consider an equivalent language over the environment of a type directed storage system [2] in which each data element carries a tag to indicate *address* or *data (literal)*.

THE LANGUAGE

In a type directed system, a *fetch* cycle is to result in the retrieval of a data element; if the tag indicates 'data' the *fetch* cycle is complete and the element is passed to its destination; if the tag indicates 'address' then the *fetch* cycle is repeated using that address as a new operand.

Similarly in a *store* cycle (operating over an operand address and a value to be stored) the determination of the address of the cell which is to receive the specified value is completed when the operand references a cell containing a data element.

The primitives of the minilanguage in this environment are:

literals	'A', 'B', 'C', ..., 'Z'
pointers	@A, @B, @C, ..., @Z
identifiers	A, B, C, ..., Z

An assignment statement is composed of a string of the form:

$$e_1 := e_2$$

where e_1 is a left hand side expression which is an identifier and e_2 is a right hand side expression which may be a literal, a pointer, or an identifier. The execution of an assignment statement is expressed by the following functions:

$execute\ (e_1, e_2) =$

$\quad store\ (eval\text{-}lhs(e_1),\ eval\text{-}rhs(e_2))$

$eval\text{-}lhs(e) =$

$\quad is\text{-}literal(fetch(e)) \rightarrow e$

$\quad T \rightarrow eval\text{-}lhs\ (fetch(e))$

$eval\text{-}rhs(e) =$

$\quad is\text{-}literal(e) \rightarrow e$

$\quad is\text{-}pointer(e) \rightarrow e$

$\quad T \rightarrow fetch(e)$

$fetch(e) =$

$\quad is\text{-}literal(e) \rightarrow e$

$\quad is\text{-}pointer(e) \rightarrow fetch(e)$

The function *store (id,v)* operates over the environment to associate the value *v* with the identifier *id*.

A program in ML2.5 is composed of a sequence of assignment statements which are to be executed in sequantial order over an environment which is initially empty.*

Examples:

1)
```
X := 'A'
Y := @X
X := 'B'
Z := Y
```

2)
```
X := 'A'
Y := 'B'
Z := @X
Y := @Z
Z := Z
```

*In a host system, this initial environment may be modified to be that of the host system.

```
3) X := @A
   Y := @X
   Z := @Y
   A := @Z
   W := A
```

```
4)  X := @A
    Y := @X
    Z := @Y
    A := @Z
    Y := 'A'
```

After executing the program in example 1, cell X will contain the literal 'B', Y will point to cell X and Z, through reference to Y, will contain the literal 'B'.

In example 2 after the execution of the first four statements cell X contains 'A' and Z contains a pointer to cell X. Thus the execution of the assignment statement Z := Z will fetch the value 'A' from the cell X by indirection through Z and deposit that value into cell X.*

Example 3 shows that cyclic addressing is possible and that a right hand side reference to an element of the cycle results in a non-terminating *fetch* cycle. Similarly example 4 contains the same cycle which is not broken by assignment of a literal to any element of the cycle. Conversely in the original minilanguage 2 may contain apparent cycles but since the number of *fetch* cycles is explicitly specified in the language (e.g., ↓↓↓B) the *fetch* cycle terminates.

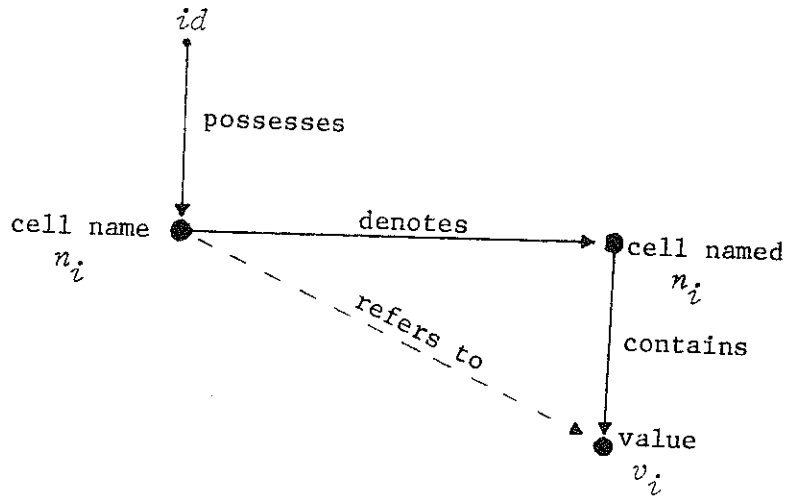
DISCUSSION

Examination of ML2.5 will reveal two disjoint sets of objects; literals and pointers. Further, examination of the *eval-rhs* function will reveal that the context of a pointer determines its influence on the interpreter over the language; on the right hand side of an assignment statement a pointer is an object to be assigned to the location

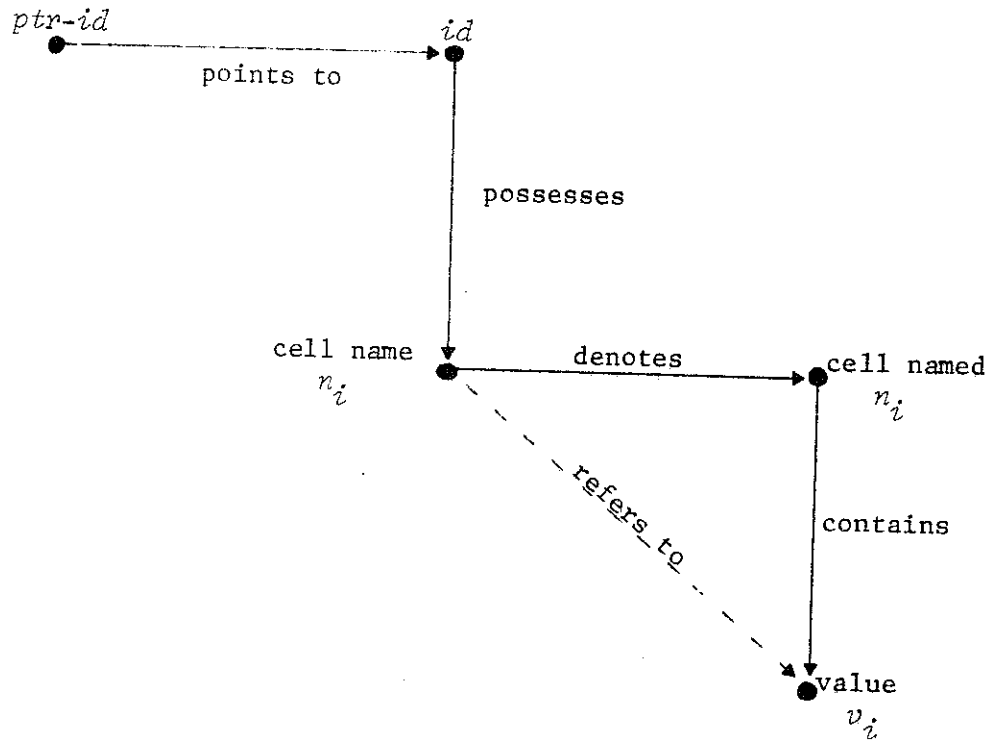
* That is, statements of the form Z:=Z are consistently "do nothing"statements.

associated with the evaluation of the left hand side. As the contents of a cell, a pointer initiates that at least a further *fetch* cycle is necessary in the retrieval of data from the storage system.

Wegner [4] has pointed out that the model of the association of a value v_i, a cell, a cell name n_i and an identifier id is represented by the diagram:



However, in this language there exists a further branch on this tree which terminates at the identifier *(id)* and originates at a pointer *(ptr-id)*. Whilst this language assumes a unique relationship from an identifier through the cell name and the cell, to a value, it must also assume that there exists only one 'points to' relationship. However the pointer value may be contained in many cells.



This same extended relationship exists in Ledgard's Minilanguage 3 (Generalized Transfer of Control).

Whilst the majority of implemented addressing schemes are nominative in nature, the IBM 1620 system had a mixed scheme. Once having specified indirection, by the flag over the low order digit of the address field, the *fetch* cycle was repeated until the selected address field contained a direct (unflagged) address. One more *fetch* then completed the cycle to obtain a data value.

Acknowledgment

The author wishes to acknowledge the constructive comments of H. Ledgard, the author of the original paper on minilanguages, and for his encouragement to offer this alternative to his minilanguage 2.

References

[1] Foster, C. C. Architects Sketch Pad: Symmetry of Addressing Modes. Computer Architecture News, SIGARCH, ACM, Vol. 1 No. 1, January 1972.

[2] Iliffe, J. K. Basic Machine Principles, MacDonald Computer Monographs, America Elsevier Pub. Co., New York, 1968.

[3] Ledgard, H. F. Ten Minilanguages: A Study of Topical Issues in Programming Languages, Computing Surveys, Vol. 3, Sept. 1971.

[4] Wegner, P. The Vienna Definition Language, Computing Surveys, Vol. 4, No. 2, June 1972.