Preliminary ZENO Language Description

J.E. Ball, G.J. Williams, and J.R. Low
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR41

January 1979

Abstract:

The specification of ZENO, a programming language intended as the target language for a research project in advanced compiling, is presented. The language is strongly based on EUCLID, with modifications for message-based parallel processing and a somewhat different treatment of data types.

This research was supported in part by the Alfred P. Sloan Foundation under Grant No. 74-12-5, and in part by the National Science Foundation under Grant No. MSC76-10825.

Preliminary ZENO Language Description

I. Introduction

The programming language ZENO is being designed to act as a test bed for research in distributed processing and in sophisticated code optimization. It does not represent a full-fledged attempt to design the perfect general purpose programming language. Instead, our goal has been to define a language which is typical of recent language designs and which includes those constructs that we feel have particular significance for research in compiler design and distributed processing. It is presented here because some of its features have applications in current attempts to define reliable system programming languages.

The basic language is in the ALGOL-60, PASCAL, and EUCLID tradition, and the bulk of the language is derived directly from EUCLID. However, it is important to note that our goals are quite different from those of EUCLID. The EUCLID designers have stated that EUCLID was "... designed to facilitate the construction of verifiable system programs" [Lampson et al., 1977]. They later state "... the language must not require a highly optimizing compiler to achieve an acceptable level of efficiency in the object program." EUCLID was also "... designed to permit one-pass translation."

We are proposing to construct a highly sophisticated optimizing compiler (of necessity multi-pass) and thus we have altered some of the design of EUCLID. We do not consider verifiability as a prime goal, but we treat assertions as a very important source of information for the compiler. Code optimizations are performed by recognizing certain relationships among the expressions and statements of a program which allow us to generate better code than could be generated without knowledge of such relationships. Assertions and their implications can provide the optimizing compiler with additional knowledge of such relationships.

ZENO is being designed with mechanisms for distributed computations [Feldman, 1976]. In particular, ZENO includes ways of talking about independent instantiations (parallel processes) of programs communicating with each other using only messages.

The other major difference between ZENO and EUCLID is in the notion of data-type. Rather than use data types to express restrictions on the range of variables (EUCLID subrange types), we use assertions to specify the properties (including value range) associated with each variable.

In the remainder of this report, we will assume the reader is familiar with EUCLID. Rather than defining the entire language, we will define ZENO in terms of the differences from EUCLID. We will separate those into two categories: constructs which have no counterpart in EUCLID and constructs which are similar in general function to those of EUCLID but different in detail.

II. Constructs Not Found in EUCLID

A. Asets and Names

An aset (think of an unordered LISP Association-List) is an aggregate data structure (much like a record) which maps symbolic names into values. It is conceptually a set of ordered pairs (called slots) of the form (name~value). The operations on an aset are: construction; putting a slot into an aset, using put; removing a slot from an aset, using remove; testing whether a slot is in an aset, using present in; and fetching and assigning to the value of particular slots. Asets provide a generalization of EUCLID records and form the basic construct used in messages.

1. Names: Names are the keywords to acquire or store values from the slots of an aset. Each name has a data-type associated with it.

name Example : sometype

would declare "Example" to be a name whose associated values are of data-type "sometype".

Names are constants. There are no variables whose values are names. If there are several declarations of the same symbol as a name, they are different objects in the standard EUCLID sense. Names imply access to slots of an aset, therefore they must be explicitly imported into any scope which references them. Currently there is a restriction that pointer types, asets, and modules; and records containing pointers, asets or modules are illegal as the type associated with a name.

2. Aset: Asets are declared in the same way as other variables (with initialization optional):

var AnAsetVar: aset

or constants (which must be initialized):

const EmptyAset : aset := empty

3. Adding Slots: Slots are added (or replaced) in an aset variable using the Put statement.

```
put slotname~value in asetvar
```

This will place the slot ("slotname"~"value") in the aset named "asetvar". If there already was a slot whose name was "slotname", that slot would be deleted.

4. Creating Aset Values: A syntactic extension allows us to construct an entire aset value at once:

```
Asetvar := (N1~V1, N2~V2,...,Nk~Vk)
```

would be semantically equivalent to:

```
[
  var temp: aset := empty
  put N1~V1 in temp
  put N2~V2 in temp
    .
    .
    .
  put Nk~Vk in temp
  Asetvar := temp
]
```

Thus, we may have a sequence of one or more (name~value) pairs.

5. Removing Slots from Messages: Removal of a slot is done using the remove statement:

```
remove slotname from asetvar
```

would remove the slot whose name component was "slotname" from the aset variable "asetvar." If there is no such slot in the aset variable, then the statement is a no-op.

6. Referring to Values of Slots: We refer to the values of slots in much the same manner as we refer to the fields of a record. We give an expression which is of type aset, a period, and then the slot name. Thus we have:

```
type actiontype = (start, stop, proceed)
name action: actiontype
var asetvar: aset := empty
put action~start in asetvar
if asetvar.action = start then print("yes") end if
asetvar.action := stop
if asetvar.action not= start then
```

```

    print("no")
  end if

```

It is considered illegal to reference a slot that does not exist. Thus, the following sequence would cause a fault:

```

asetvar := empty
asetvar.action := proceed

```

That is, assignment to the value component of a slot does not create the slot.

7. Testing the Presence of a Slot: Performed using the present construct which evaluates to the boolean value true if the name given is that of a slot within the aset expression, and false otherwise.

```

  if Action present in actionvar then
    actionvar.Action := proceed
  else
    put Action~start in actionvar
  end if

```

8. Aset Assignment: Assignment has a value semantics; that is, if we have

```

  put action~start in asetvar1
  asetvar2 := asetvar1
  asetvar1.action := proceed

```

and we were then to test

```

  asetvar2.action = start

```

the result would be true. There is no concept of sharing between asets.

B. Parallel Execution of Modules

We wish to allow parallel execution of independent processes. To describe these processes we will use module definitions with certain restrictions.

1. The module may not import variables in any form. Only constants, types, names of top-level constant processes, and slot-names may be imported.

2. The handle of a particular module is called a moduleId (for module identifier). A data type has been added for such values. Assignment and comparison for equality are the

only operations defined for moduleId values.

```
var MOD1 : moduleId
```

3. An instance of the module is created using the fork statement, which assigns the moduleId of the newly created process to a variable:

```
MOD1 := fork moddef
```

where "moddef" is the name of a module type satisfying restriction (1). The module instantiation may have actual parameters.

4. The body of the process is contained in the initially clause.

5. The lifetime of a process is not dependent on its creator. A process may only die when it terminates itself (by exiting the initially clause or executing a self destruct statement), or if the system determines that the process is no longer accessible (no other process has its moduleId, and it is in a receive state). In either case, the finally clause (if any) of the module will be executed on the destruction of the process.

C. Messages

Zeno processes communicate with each other by means of messages [Feldman, 1976]. A Zeno message is an aset expression to which "Sender" and "Transaction" slots are added. The Sender slot is of type moduleId and is set by a send operation to equal the moduleId of the sending process. The Transaction slot has type "transaction", a Zeno primitive type which is used for control of message interactions. Assignment and comparison for equality are the only defined operations for transaction values.

```
var TRAN1 : transaction
```

New values of type transaction are created by the expression "new transaction" which generates a unique transaction value:

```
TRAN1 := new transaction
```

A message is sent to another process by means of the send statement:

```
send ASET1 to MOD1 about TRAN1
```

The "about transaction" clause is optional and if omitted the statement is equivalent to:

send ASET1 to MOD1 about GeneralTransaction

where "GeneralTransaction" is a system defined pervasive transaction constant.

Messages are queued for each destination process and can be received using the receive statement, which specifies a set of acceptable values for the Sender slot in the received message and similarly, a set of acceptable Transaction values. The from and about clauses are both optional and if omitted any value for the corresponding slot will be accepted. The receiving process is suspended until the arrival of an acceptable message, which is then copied into the specified aset variable. As in [Feldman, 1976], all messages which are sent to the same process and transaction from a given sender are guaranteed to arrive in order.

receive ASETVAR1 from (MOD1, MOD2) about (TRAN1, TRAN2)

Processes can test for messages waiting to be received by means of a boolean expression similar to a receive construct as follows:

if pending from (MOD1,MOD2) about (TRAN1) then
 .
 .
end if

D. Data Typing

Here we make a significant departure from EUCLID. The basic change is that we have eliminated subrange types - their functions in EUCLID are duplicated and extended by means of "properties" of variables and by subrange expressions.

Named subrange constants may be declared in ZENO as follows:

const OneToHundred = 1..100
const citrus = grapefruit..orange

Subrange expressions may be used in the specification of sets, in the specification of array index bounds, in boolean expressions testing interval membership (expression in subrange), as a case label in the select statement, and as the generator of a for loop.

Subranges may not be used to declare variables in Zeno. However, we can get some of the same effects using properties (for a complete discussion of properties, see TR28 [Feldman & Williams, 1978]).

A property is defined for variables of a given type (simple, record, or module). There are basically two kinds of properties: predicates involving only components of the variable with that property; and uninterpreted properties which are essentially enumerations.

1. Properties as Predicates: Essentially all we must do is define a predicate. For example, the EUCLID declarations

```
type smallpos = 1..15
var x: smallpos
```

could be equivalently written in ZENO as

```
property(integer) smallpos2(z) = z in 1..15
var x: smallpos2 integer
```

Properties, though, give us much more descriptive power than types. For example, if we wanted a certain variable P to represent a point in the complex integer plane that wasn't more than five units away from the origin, we could write

```
type complex = record
                var xpos : integer
                var ypos : integer
            end record

property(complex) close(t) =
    ((t.xpos * t.xpos) + (t.ypos * t.ypos)) <= 25

var C1,C2: close complex
```

The only obvious way in EUCLID to get the same effect would be to have a module for complex with an invariant similar to our predicate "close". But this isn't totally satisfactory either. Let us also have complex numbers greater than five in magnitude.

```
property(complex) far(t) =
    ((t.xpos * t.xpos) + (t.ypos * t.ypos)) > 25

var P1,P2: far complex
```

Now consider a function to add two complex numbers together.

```
function complexAdd(a,b: complex) returns complex =
[
    var sum: complex

    sum.xpos := a.xpos + b.xpos
    sum.ypos := a.ypos + b.ypos
    return sum
] complexAdd
```


The checking of compatibility of the formal parameters with the actual parameters in ZENO is done by verifying that the base types are the same, and that the properties required of the formals are satisfied by the properties of the corresponding actuals.

Thus, in ZENO, it would be perfectly legal to say:

```

      C1 := complexAdd(C1,C2)
or
      P1 := complexAdd(P1,P2)
or even
      C1 := complexAdd(P1,C2)

```

In each case above, checking may be required to verify that the result of complexAdd satisfies the properties of the variable being assigned to. This check is similar to the check necessary for the following assignment in EUCLID:

```

var x: OneToHundred
      x := x+1

```

In EUCLID, to get the same degree of "type" checking, we would either have to construct eight different complexAdd routines (taking the various combinations of arguments and result types), or we would have to use multiple coercions, e.g.,

```

C1 := smallComplex<=<complex(
      complexAdd(
        complex<=<largeComplex(P1),
        complex<=<smallComplex(C2)))

```

Variables may have more than one property:

```

property(complex) realneg(t) = t.xpos < 0
var XX: realneg close complex

```

2. Uninterpreted Properties: The other form of property is just an identifier to which the user has given a special meaning. Its major use is to allow the programmer to use (and have checked) properties or attributes which either do not depend on the value of the variable (such as in dimensional analysis) or cannot be defined in the assertion language provided (such as the graph property "tree"). An example of the use of properties for dimensional analysis is:

```

property (integer) meters
property (integer) centimeters
property (integer) inches
property (integer) squareCentimeters
property (integer) squareInches

```

```

var x,y: centimeters integer
var q,z: inches integer
var xx,yy: squareCentimeters integer
var qq,zz: squareInches integer
var E: meters integer

```

Then the compiler can check that we are not assigning inches to centimeters or square inches to inches. Thus, the following statements would be illegal:

```

x := z
q := qq

```

To allow arithmetic operations on variables with uninterpreted properties we must provide propagation rules.

propagation rules

```

inches - inches => inches
inches + inches => inches
inches * inches => squareInches
centimeters * centimeters => squareCentimeters

```

Thus, the following statements would now be legal:

```

xx := x*y
y := x+y

```

To perform uninterpreted property conversions, we can use a cast operation:

```

E := (meters: 30)
x := (centimeters: E*100)

```

A set of mutually exclusive properties may be defined as an enumeration:

```

property (integer) color = (red, blue, yellow)

```

This declares the properties red, blue and yellow, as well as the property color which is defined to be their exclusive-or. An integer may have at most one of the properties red, blue or yellow.

We can also define a property in terms of previously defined properties:

```

property (integer) warm = red or yellow

```

The assume statement is used to declare relationships between uninterpreted properties. It provides the compiler with inferences which it can use when checking property correctness. The assume statement may contain implications between simple boolean expressions containing only property

names and types. The declaration of warm above is equivalent to:

```
property (integer) warm
assume red or yellow <=> warm
```

Another example of an assume statement would be:

```
assume tree => acyclic
```

E. Minor Additions

1. Block Labels

Blocks and loops may be labeled with a string constant, e.g.,

```
loop 'name'
.
.
end loop 'name'
```

or

```
[ 'name'
.
.
.
] 'name'
```

These labels are useful in keeping track of block nesting and for the extended exit and continue statements.

2. Exit and Continue Statements

Exit statements may take the name of the block (or loop) to be exited, and the continue statement may be used to proceed to the next iteration of the (optionally specified) loop.

```
loop 'outer'
.
.
loop 'inner'
.
.
if test then
  continue 'inner'
else
  exit 'outer'
```

```

    end if
    .
    .
    end loop 'inner'
end loop 'outer'

```

3. Keyword and Optional procedure parameters

ZENO extends the EUCLID syntax for routine parameters to allow actual parameters to be specified by their formal parameter names as well as positionally. Any prefix of the parameter list may be specified by position; additional arguments can be provided using the syntax formalParameterName~value. In addition, formal parameters may be marked as optional and given a default value. Such parameters need not be supplied in every call to the procedure.

```

[
  procedure ReadNumber(var value: integer,
    optional prompt: String(parameter) := 'Value: ',
    optional radix: small integer := 10 ) =
  [
    .
    .
    ] ReadNumber
    var x,y: integer
    .
    .
    ReadNumber( x, radix~8 )
    .
    .
    ReadNumber( prompt~'Next value = ', value~y )
  ]

```

4. Extended loop syntax

ZENO includes explicit syntax for while and until loops as follows:

```

while boolexpr
loop
  .
  .
end loop

loop
  .
  .
end loop
until boolexpr

```

Also, a special syntax is provided to facilitate the expression of common "search loops" (similar to that in Alphard [Hilfinger et al., 1978]); in which a test is evaluated for each element generated and a statement is executed for the first true occurrence or, if none is found, a second statement is executed.

```

find first i in 1..10 such that A(i)=4 then
.
.
otherwise
.
.
end find first

```

ZENO also allows a 'such that' clause in the EUCLID for loop syntax:

```

for i in 1..100 such that A(i) > x
loop
.
.
end loop

```

which is equivalent to the EUCLID sequence:

```

for i in 1..100
loop
  if A(i) > x then
    .
    .
  end if
end loop

```

5. Extended select (case) statement

ZENO includes a modified and slightly extended version of the EUCLID case statement which allows transfers of control between cases. The do case command within a case element causes a control transfer to the newly specified case element.

```

select a*5 of
[
  case 2..10:
    . . .
  end case

  case 11:
    . . .
    do case x
  end case

  default:

```

```

      . . .
    end case
  ]

```

III. Roughly Equivalent Features

A. Assertions

The assertions in ZENO are of two forms: the point assertion and the range (under) assertion.

1. Point Assertions

A point assertion is simply a boolean test (without side effects):

```
assert boolexpr
```

If the expression is true the program will continue; if false the program will halt with an assertion violation. If the compiler cannot infer from an analysis of flow that the assertion is true, a runtime check will be generated.

Note: Immediately after a point assertion, the boolean expression asserted is true.

An important aspect of the optimizations performed by the compiler will be to minimize the runtime expense of assertions either by being able to prove the assertion is true (thus eliminating any runtime checks) or by generating the runtime check using minimal overhead (taking advantage of traditional program transformations, especially code motion). Assertions also provide hints to the compiler about which information is useful to keep during flow analysis.

2. Under Assertions

Often we wish to have an assertion be true over a range in the source program (such as a loop invariant). To do this we will label a block with an under assertion. This will act the same as if we had placed a point assertion before and after each statement textually enclosed in the labelled block, e.g.,

```

[ under a(i) <= a(j)
  x := y
  procedurecall(a,i,j)
  if i not= j then
    a(i) := 0

```

```

        a(j) := 5
    end if
}

```

This would act the same as if we had placed the point assertion

```

assert a(i) <= a(j)

```

before the first statement and after each statement enclosed in the block.

Note: For a routine call the assertion need only be true before and after the call. Nothing is specified about the assertion during the execution of the procedure (unless of course the procedure body itself has the same assertion).

3. Relax

Occasionally we wish to relax an 'under' assertion (not require it to be true) for a portion of a block. To do this we will label a subblock with a relax clause:

```

[ under odd(x)
.
.
[ relax odd(x)
  x := x+1
.
.
  x := x+1
]
.
.
]

```

4. Naming Assertions

Sometimes it is convenient to name an assertion (for example, if it is to be referenced in a subsequent relax). Named assertions may be declared as follows:

```

assertion a1: x < y
assertion a2: x < y-1
[ under a1
.
.
[ relax a1
.
.
]
assert a2
.
.

```

```

    ] assert a2

```

The scoping rules for assertion names are the same as those for any other identifier. Pervasive may be used to cause the assertion name to be inherited by inner closed scopes:

```

pervasive assertion a3: x < y

```

B. Aliasing

In EUCLID it is very important that two identifiers do not stand for the same variable and steps are made to outlaw this. In ZENO, we will assume that two identifiers are not aliases. However, for those special cases where aliasing is potentially useful, we will allow the user to tell the compiler by relaxing the NoAlias assumption:

```

procedure mergearray(var dest,src1,src2:
    array 1..256 of integer) =
[
    relax NoAlias(dest,src1,src2)
    .
    .
]

```

This will allow (but not require) mergearray to take the same array as the actual parameter corresponding to two or three of the formal parameters.

C. Sets

It is important for dealing with messages that we can have sets of transactions or sets of moduleIds. We also want sets of other data types. Therefore, ZENO allows sets of any type except modules, asets, or records, e.g.,

```

var abc: set of moduleId

abc := (mod1,mod2,mod3)

```

The operations on sets are the same as in EUCLID:

+	union
-	set difference
xor	symmetric difference
*	intersection
in	membership
<=, >=	inclusion

etc.

Preliminary ZENO Language Description Bibliography

Feldman, J.A. "A Programming Methodology for Distributed Computing," TR9, University of Rochester Computer Science Department, September 1976.

Feldman, J.A. and Williams, G. "Some Remarks on Data Types," TR28, University of Rochester Computer Science Department, April 1978.

Hilfinger, Paul; Feldman, Gary; Fitzgerald, Robert; Kimura, Izumi; London, Ralph L.; Prasad, K. V. S.; Prasad, V. R.; Rosenberg, Jonathan; Shaw, Mary and Wulf, William A. "An Informal Definition of Alphard (Preliminary)". Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa, February 1978.

Jensen, K. and Wirth N. Pascal User Manual and Report, 2nd Edition. Springer-Verlag, 1974.

Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. L. "Report on the Programming Language Euclid." SigPlan Notices, 12, 2 (February 1977), 1-79.

Liskov, Barbara; Moss, Eliot; Schaffert, Craig; Scheifler, Bob; and Snyder, Alan. "CLU Reference Manual." Massachusetts Institute of Technology, Laboratory for Computer Science, Computation Structures Group Memo 161, July 1978.

Example Zeno Program

```

pervasive type ProcessManager = module
imports
(
  name
    command: commandEnumeration
    key, entry, member: moduleId
    entriesFound: set of moduleId
    notify: moduleId
)

procedure NotifyAll(msg: message) =
[
  for p in msg.entriesFound
  loop
    { this may send to processes that are dead ! }
    send (command~death, process~msg.process) to p
  end loop
] NotifyAll

initially
const
  { HashTable and Set are assumed to be pervasive
    module types (i.e. prototypes) }
  InterestedProcs: moduleId := fork HashTable(1000)
  AliveSet: moduleId := fork Set(100)

var
  msg: message

loop
  { messages are of four kinds: replies from InterestedProcs,
    replies from AliveSet, messages from the system concerning
    the birth and death of processes, and messages from
    processes asking to be notified if another process dies. }

  msg := receive

```

if msg.Sender = InterestedProcs then

{ replies from InterestedProcs are replies either to an
"enter" request ("entered" or "overflow"), or to a
"findEntries" request ("found" or "notFound"). }

select msg.command of

case overflow:

Abort('Process manager hash table overflow')

end case

case found: { notify all interested processes }

NotifyAll(msg)

end case

case notFound:

{ no one was interested in the death of this
process, so no action is required. }

end case

case entered:

{ a new entry has been added to the table }

end case

end select

elseif msg.Sender = AliveSet then

select msg.command of

{ replies from AliveSet are replies to "addMember"
("entered"), "removeMember" ("removed"), or
"findMember" ("found" or "notfound"). }

case found:

{ We received a pleaseNotify message, so a
findMember message was sent off to AliveSet to
check that the process in question was not already
dead. The reply "found" indicates that the process
is still alive, so the pleaseNotify request is
entered into the table. }

send (command~enter, key~msg.process,
entry~msg.notify) to InterestedProcs

end case

case notfound:

{ Same as above, but the reply "notFound" indicates
that the process is already dead, so a "death"
message is sent off to the interested process. }

send (command~death, process~msg.process)
to msg.notify

end case

case entered,removed:

end case

end select

```

else      { message is from someone else }

  select msg.command of

    { the following messages are unsolicited, and
      trigger a sequence of messages.  }

    case death:
      send (command~findEntries, key~msg.process)
        to InterestedProcs
      send (command~removeMember, member~msg.process)
        to AliveSet
    end case

    case birth:
      send (command~addMember, member~msg.process)
        to AliveSet
    end case

    case pleaseNotify:
      send (command~findMember, member~msg.process,
        notify~msg.notify) to AliveSet
    end case
  end select
end if
end loop
end initially

end ProcessManager

```