2

Comparison of Programming Paradigms

2.1 Machine States

2.1.1 Procedural Languages

A computer is essentially a finite state machine, though the number of states is astronomical. CPU registers, primary memory, mass storage and so forth record the machine state. A computer program is a sequence of instructions that controls the changes of state. The execution of machine instructions makes the machine change to a new state which is dependent upon the previous state.

Procedural languages closely reflect this primitive machine level. A "*procedural variable*" is a symbolic or logical abstraction from machine state. The values of variables are a record of current machine state. Each program instruction is an explicit description of a machine state transition. Statements of high-level procedural languages hide the detailed information of low-level state transition. For instance, the assignment: "a = 1", has transitions: "load 1 into register", "store register to location a". The compiler translates the high-level instructions to machine instructions. (Thus, high-level procedural programs explicitly specify the *logical* state transitions instead of the machine-level state transitions. To be precise, we use the term "machine state" to refer to the "logical state" rather than the actual machine-level state.)

Because each state transition is dependent upon the previous state, it is sensitive to the order of execution of the instructions. The programmer must explicitly specify a precise order of the instructions. Consider the following **BASIC** program:

```
100   a=1
110   b=2*a
120   sum=a+b
```

The **BASIC** statements must be executed in the pre-defined order (indicated by line numbers); otherwise they will get an unintended answer for sum. If, for

example, lines 100 and 110 were swapped, sum would have the value 1 rather than 3 (note that all variables in **BASIC** are initialized with 0).

2.1.2 Functional Languages

Functional languages hide the machine states by abstractly viewing a computation as the result of evaluating a mathematical function. An application program is a function built up from simpler functions. Each function has its own local data and functions. Since all functions are communicated through parameters and the values returned by the functions only, the major advantage is the elimination of interference among functions. Without interference, which is a major barrier in analysing data flow in programs, the synchronization of evaluation of functions is handled by the functional language compilers. All intermediate machine states are unknown to the programming. Hence, the programmers can only specify two possible states: the input parameters (initial state) and the output (final state).

On the other hand, this abstraction imposes a great restriction on the applications. Since pure functional languages do not have variables, the result of a function call cannot be stored and re-used in a later session without re-evaluating the same function.

In a functional language such as **miranda** [RSL87], a script defines the function and the initial condition (also represented by functions). For instance, a **miranda** script:

```
sum = a + b
    where
        a = 1          || comment: initial condition
        b = 2 * a
```

The function sum is at the highest level of abstraction of the application. It consists of functions a and b. If the user wants to compute sum for a = 2, he/she must edit the entire script of sum to change the definition of a and re-load the script, i.e. re-run the application (unless a was specified to be a parameter of sum). This *edit-run* cycle is natural in the development stage of the program, but obviously unacceptable for serious applications.

Some functional languages, such as **Lucid**[1] [AW77;WA85], regard input and output of functions as infinite streams of data. A function filters the input streams to generate the output streams. Thus, the definition of a, in the above example, becomes an input stream whose values can then be modified by external factors:

```
sum
    where
        sum = a + b
            where
                a = 1 fby input
                b = 2 * a
            end;
    end;
```

where fby (followed by) is an data-flow operator that merges two streams to form a stream whose first element is the first element of the left operand (in this case, an infinite stream of 1's), and the rest of the stream is taken from the second stream (input). **Lucid** treats any free variable, such as the variable input in this example, as user input; **Lucid** will prompt for input data. However, **Lucid** does not allow incremental modification of the functions. There is no way to alter the definitions, for instance, to change b to x * y + z, without re-doing the *edit-run*.

Some functional languages, such as **SML** [Ha86], allow dynamic modification of functions. However, a modification may introduce interference problems (as in procedure languages). For example, in the following **SML** program:

```
fun f() = 1;
fun g() = f();
fun f() = 2;
```

What does the function g evaluate to (1 or 2)? In the case of **SML**, g evaluates to 1 because f() was evaluated when g was being defined. Such method of binding was called *static binding* or *early binding*. In other functional languages (or even other implementations), the result may be 2 if the binding was *dynamic binding* or *late binding*; i.e. the r.h.s. of g's definition would not be evaluated unless g is evaluated—thus, function g is bound to "a function that evaluates f" rather than "the (current) result of f()".

---

[1] Lucid is also known as a data-flow language.

The semantics of functional languages does not specify clearly which interpretation is appropriate. It is the implementation of the compiler or interpreter which determines the strategy of evaluation.

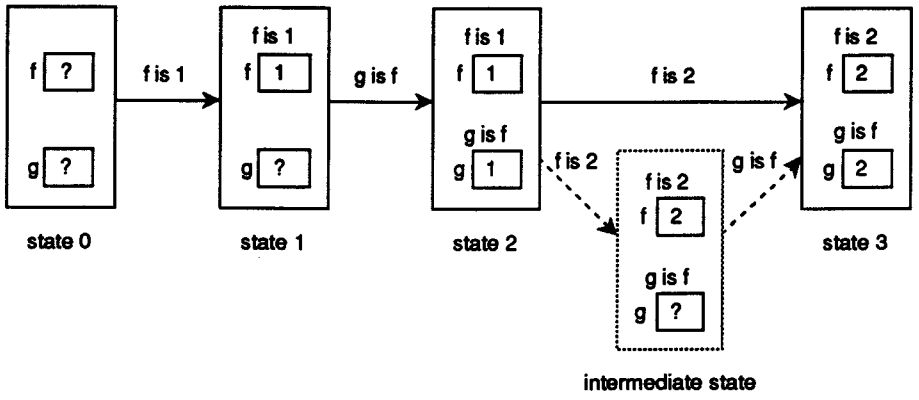
2.1.3 Definitive Languages

Definitive languages, such as EDEN, in contrast to functional languages, guarantee the consistency of definitions. Thus the following EDEN definitions:

```
f is 1;
g is f;
f is 2;
```

guarantee that g has the same value of f. So, when f was 1, g had the value 1; when f becomes 2, the value of g changes to 2. (EDEN interprets the program statement by statement. When the same variable is re-defined, the last definition overwrites the previous one if the new definition doesn't introduce cyclic dependency.)

Here, the values of variables f and g represent part of the machine state, and the set of definitions implicitly specifies the state transition. It is important to note that the definitions are, themselves, *part* of the machine state because the data dependency information is, somehow, extracted from them so that the definitions can be maintained.



Definitions can be freely modified throughout the computation process. The third definition, "f is 2", overwrites the previous definition, "f is 1". This transition of state is composed of many intermediate state changes, such as "evaluate f" and "evaluate g". These intermediate state changes are governed by the current set of definitions (see the figure above). On this account, definitive languages, unlike procedural languages, do not require a precise order of definitions to achieve the correct final state. Hence the set of definitions can

be regarded as a *specification* of the final state rather than a sequence of instructions to generate it. The three EDEN programs below produce the same final state although each of them may have different intermediate states.

a **is** 1; b **is** 2 * a; sum **is** a + b;	b **is** 2 * a; sum **is** a + b; a **is** 1;	sum **is** a + b; b **is** 2 * a; a **is** 1;

In fact, all the six permutations of these three definitions produce the same final state: a=1, b=2, sum=3.

The definitions retain the mathematical abstraction of the application. Because definitions, unlike functions in functional language, are parts of the machine state, they are free to be modified as part of program execution. However, a modification of definition is a complex state transition. This is a significant difference between definitive languages and procedural languages. Such a (definitive) machine is no longer a simple procedural machine; its control logic is sophisticated enough to organize machine instructions to produce a final state according to the definitions given.

To make it simple to understand, we may imagine that a *definitive coprocessor* runs in parallel with the *procedural processor*. This definitive processor always looks for any modification of a definition. If it detects such a modification, it will insert instructions to the procedural processor in order to maintain the specification imposed by the definitions.

2.2 The Human/Computer Interface

Humans communicate with computers through a number of I/O devices. A video display unit is a common output device. The screen display reflects the machine state to the user. Thus the screen is part of the machine state.

Procedural programmers explicitly write step-by-step instructions to draw the screen. The programmers can change the screen state in the middle of computation to reflect the intermediate state since all the state transitions are under their control.

Functional programmers are less fortunate. Since there is no representation of intermediate state at all, (pure) functional language programmers have many difficulties in producing a screen display to reflect the intermediate computation results which are important in many applications, such as CAD. Hence, I/O is a difficult area for the functional languages.

Definitive languages, in principle, can produce a screen display for each (logical) state. In fact, the screen state can be described by definitions. For example, DoNaLD can automatically update the screen state to reflect the abstract geometric objects described by the definitions.

In practice, DoNaLD has no problem in mapping each geometric object onto the screen, but has difficulties in expressing covering relationships between the images, i.e. the graphical images may overlap in arbitrary order. The problem is dominated by two factors: the DoNaLD notation itself, and the nature of the screen devices or the interfaces:

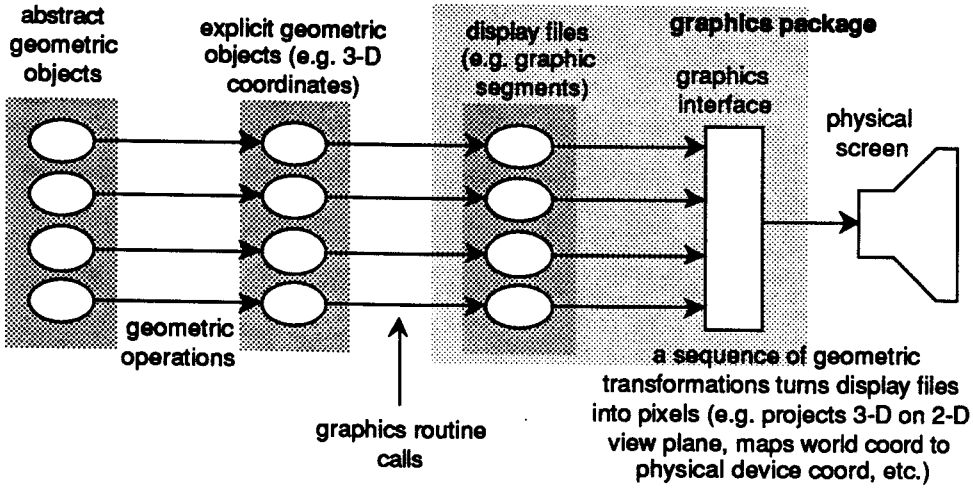
- The current DoNaLD interpreter implements a two dimensional subset of the original notation [BABH86]. This subset has no algebraic information to describe the covering relationships between objects. A full 3-D notation can easily solve the problem. Other similar problems reflect the deficiency of the language rather than the programming paradigm.

  In contrast, the disposition of text blocks can be expressed in **SCOUT** notation, a preprocessor of EDEN [Yu88]. **SCOUT** translates high-level text screen layout definitions into simple EDEN definitions, and the EDEN interpreter can hence format text strings into a number of text boxes which can be overlapped as specified by the **SCOUT** definitions.

- Video display devices, usually, require sequences of control signals (e.g. the escape sequences on text terminals) to control the screen layout (e.g. the position of a character). This procedural nature of the device makes the use of a procedural interface unavoidable. Because the definitive machine changes the machine state in an order that the programmer cannot control, the compatibility between the procedural interface and the definitive machine is crucial. For instance, the **SunCore**[SUN88b] graphics interface allows one graphic segment to be opened at a time; an opened segment must be closed before another segment can be opened. If two graphics objects are drawn at the same time, i.e. two draw procedures are interleaved by the definitive machine, the interface fails. However, if the interface has no such restriction on the creation of graphic segments or the definitive machine does not interleave drawing procedures, the interface and definitive machine are compatible. The current DoNaLD implementation employs the **SunCore** package and a non-interleaving definitive machine. Similar

problems in dynamically modifying objects using graphic segments are discussed in [SDS88].

The output device cannot directly reflect the literal machine state, which is represented by an astronomically large dimensional data set. Only some selected data is projected or mapped onto the output, and this data is transformed, in certain ways, to human understandable forms in the graphics pipeline.



Functional languages are good at formulating geometric transformation and projection functions, but poor in control (this is also true for other declarative programming paradigms); procedural languages have better control on applying the projection, but are less effective in the implementation of the projection functions. Definitive languages lie in between. *Could definitive languages be the more appropriate compromise?* This is not an easy question to answer. It requires more research on human/computer interfacing beyond the scope of this thesis. However, through the experience of the DoNaLD project, we may gain more insight.

The EDEN language integrates both procedural and definitive paradigms. Procedures can be defined in EDEN to interface with I/O devices. These procedures (in EDEN terms, they are called *actions*) can be made under the control of a definitive machine (definition manager) without losing the generality of definitions. We shall explain the details of EDEN in the rest of this thesis.

2.3 Definitive Programming versus Functional Programming

The major divergence of definitive languages from functional languages lies in the treatment of variables. Some pure functional languages don't even have

variables. The main advantage of omitting variables in the language is to eliminate undesirable side-effects. Hence, many functional languages restrict the access of variables (e.g. variables can only be defined once) or totally prohibit the use of variables. On the other hand, as discussed in previous sections, without variables, it is difficult to express (current) machine state which is important for some applications. (A comparison of the programming concepts in imperative and applicative paradigms can be found in [GJ87].)

In contrast, variables, more precisely *definitive variables*, are the core of definitive languages. We do not initially assign an explicit value to a definitive variable; instead, we associate an implicit formula with each variable. Definitive languages keep track of the data dependency among variables implied by the formulae and update the values of variables whenever it is necessary. The term *definition* refers to the variable and the associated formula collectively.

Since definitions are usually expressed as functions of values of variables (which may in turn be specified as definitions), one may confuse these two programming paradigms, definitive programming and functional programming. For example, the definition notation, **ARCA**, [Be86a] does not have any automatic visual feed-back mechanism,[2] an **ARCA** program *appears* more like a functional language program with some special geometric operators.

The advantage of having dependency among variables is illustrated by the DoNaLD notation. Like **ARCA**, DoNaLD objects are defined by functional formulae, but an automatic graphical feed-back mechanism is also implemented. Therefore, the user can immediately see the effect of a modification of some definitions without giving a draw command explicitly. In other words, variables play a much more *active* role than those in functional languages or other conventional languages; a variable is not only a passive data holder but also an active object which knows how to refresh the data with some limited knowledge—the associated formula. These active objects are collectively modelled by the definitive machine (definition manager).

More importantly, the effects of a modification are incremental. This means that only those variables dependent upon the data which is being modified shall be updated. The definitive language can simply retrieve the value stored in a variable without re-evaluating the associated formula if no recent change has

---

[2] The user has to give a command to draw the object. Thus, the user can view the selected part of drawing with different attributes.

been made to the values of the variables on which it depends. A functional approach, on the other hand, must evaluate all the related functions to get the answer.

To be fair, definitive languages do not necessarily implement this *memorisation* mechanism. But this mechanism can be easily implemented with the cooperation of the data dependency maintainer. Some functional languages implement some sorts of memorisation using a history which records the results and parameters of recent function calls. If a function call is issued, the system first checks the parameters against the history list. If there is a match, it uses the result stored in the history rather than evaluating the function again.

Another limitation of functional languages is the restriction on re-definition of (parts of) functions due to data abstraction.

Despite these differences, definitive languages are quite similar to functional languages in their appearance.[3] There is no reason why we cannot implement definitive languages using functional languages, but neither is there any overriding advantage in using functional languages as the underlying system. Due to the inefficient implementation of current functional languages, the dynamic analysis of data dependency may be too slow to be practical. (In [CS86], some experiments were carried out to assess performance of the implementation of **KRC**, **LispKit** and **Prolog**.)

2.4 Definitive Programming versus Logic Programming

Relationships between objects can be represented by facts or predicates stored in the database of a logic programming language, such as **Prolog** [Na86; Cr88]. Logic languages have no (global) variables but a database. In **Prolog**, we can assert and delete predicates in the database dynamically. So, we can use predicates to record machine state.

However, logic languages do not provide any built-in mechanism to maintain these relations (data dependency). The programmer has to write a program to analyse the dependency by searching the database. It is obvious that the dependency maintenance job is a time consuming process. The inference engine used by most logic languages may be less efficient than the traditional methods.

---

3 Definitive languages have less restrictions on side-effects. But this difference is minor since side-effects, such as graphics drawing, are usually built into the operators. In addition, definitive languages do not specify or emphasise the use of side-effects.

(In Chapter 5, we describe some possible ways of maintaining data dependency in a definitive system.)

Logic predicates can express a richer set of relations than definitions. Definitive languages restrict the relations to be uni-directional. On the other hand, definitive languages can be implemented more efficiently.

The schemes presented in this thesis require every variable to be *strongly linked*[4] but the related definitions can be *triggered*[5] efficiently at the expense of memory. In addition, this data dependency information is important in detecting parallelism automatically. [Ba87] analysed the potential of automatically detecting parallelism in current programming languages, including imperative and declarative languages.

The definitive languages described in this thesis, such as DoNaLD and EDEN, do not incorporate generic definitions. Though we could derive instances from generic definitions by using macros or a preprocessor, it is preferable to integrate object-oriented programming concepts [Ma86] into definitive languages. Those concepts, including object class, inheritance, and message passing, found in some object-oriented languages, such as **SmallTalk** [GR83; KP86], **C++** [St86] and **HyperTalk** (Apple **HyperCard**'s underlying language) [Go87], may be used to enhance the definitive languages. In chapter 8, we propose some object-oriented extensions of EDEN.

2.5 Other Related Works

2.5.1 Implementation Techniques

In EDEN, a modification of a variable causes all the related definitions to re-evaluate. This is analogous to *data-driven* in data-flow programming terms (c.f. **Lucid** [AW77; WA85]). We may implement *demand-driven* technique [PA85; PA86]. That means related definitions will not evaluate unless there are some *demand*s upon their values. A demand is, for example, a query. However, we will concentrate on the data-driven methods in this thesis.

Definitive languages, such as EDEN, provide a dynamic analysis of the data dependency (i.e. according to the change of data dependency) to minimize the

---

4 A link represents a data dependence. Each variable is doubly linked to the directly related variables. See §3.1.3 for graphic representation of dependency link.

5 The word *trigger* means *to evaluate those related definitions.*

number of definitions needed to be computed. A definition manager in the EDEN interpreter is responsible for analysing the dependency and to schedule the evaluation. Three definition management schemes are proposed in this thesis. One of them was implemented on an EDEN interpreter and has proved to be reasonably efficient.

For a *static* set of definitions,[6] this analysis can be done at compile time on different but fixed dialogues—sets of variables that can be assigned values (with no associated formulae)[7] by the user at run-time. The result would be a conventional program running efficiently on von Neumann machines. [Mi89] presented a translator which translates a *static* set of (subset of) EDEN definitions to a **Pascal** program. However, such a translation is unlikely to be suitable for a *dynamic* set of definitions which requires dynamic analysis of data dependency.

2.5.2 Constraints

Definitions can be considered as uni-directional persistent relations. More general persistent relations are known as *constraints* [SS79; St80; Le85; La87]. Many experimental constraint-based systems have been constructed or are being developed. One of the early constraint systems, **Juno**: a constraint-based graphics system [Ne85], uses numeric methods to solve constraint equations. Other constraints systems, for example, **MetaFont** [Kn79], **Sketchpad** [Su80] and **Ideal** [Va81], use different numerical methods to solve constraint equations. The numeric methods restrict the constraints to be linear equations. Other constraint systems use more sophisticated theorem proving techniques which can solve non-linear equations as well. **Bertrand** [Le87] uses *term-rewriting* method [Bu83]—a rule-based technique (see [Ve88] for a discussion on rule-based systems). **CLP** [He86] and **CS-Prolog** [KOM87] extend logic programming to build constraint satisfaction systems. (In [FP88], there is good account of a basic classification of constraints).

The dependency maintainer used in definitive systems is simple and efficient. On the other hand, most of the constraint satisfaction systems employ relatively complicated constraint solving techniques. **Coral** [SM88] takes a uni-directional constraint approach (similar to definitive approach) to building a

---

6 This means that the set of definitions will not be modified by the user at run-time.
7 In EDEN terms, this kind of variable is called read/write variable. See chapter 3.

graphical user interface as opposed to [BD86] which takes the general constraint approach on building user interfaces.

2.5.3 Data Dependency and Qualitative Reasoning

In EDEN, definitions can be used to express predicates of the form:

if ... then ...

For example:

```
if (distance(circle1, circle2) > radius(circle1) + radius(circle2)) then valid=true
```

can be expressed as the following EDEN definition:

```
valid is distance(circle1, circle2) > radius(circle1) + radius(circle2);
```

These definitions can be used to perform reasoning because we are able to trace the information through the data dependency graph. The EDEN interpreter keeps definitions up-to-date, and it can be programmed using actions to report to the user when some conditions are violated; for example:

```
proc report: valid { if ( ! valid) writeln("**** not valid ****"); }
```

EDEN's query command can retrieve the data dependency information of a variable. So, if we find *valid* to be false, by querying about *valid*, we then know that *valid* depends upon *circle1*, *circle2*. We can then query about either *circle1* or *circle2*, and so forth, until we are satisfied.

We can also use action to maintain the truth of *valid*, for example,

```
proc if_not_valid: valid        /* an action that depends on the value valid */
{
    if (! valid)                /* if not valid */
        circle1 is ... ;        /* re-define circle1 to make the condition valid */
}
```

the action *if_not_valid*, re-defines the definition of *circle1* to keep *valid* true if necessary. It is possible for us to program *if_not_valid* to make more decisions.

Such a use of EDEN to describe and manipulate logical relationships raises issues similar to those addressed in qualitative reasoning [Bu83; Al84; Br88]. For example, the authors of [GS88a] and [GS88b] suggested a *possible worlds* approach where the *nearest world* (a set of world's status expressed in first-order predicate calculus) is the result of an action (expressed as *domain*

constraints; don't confuse with EDEN's action) acting in the *current world.* For instance, the current situation of the world is:

$$\text{on(john, chair)} \qquad \text{— (2.1)}$$

and the associated domain constraints are:

$$\text{on}(x, y) \wedge y \neq z \quad \rightarrow \quad \neg\text{on}(x, z) \qquad \text{— (2.2)}$$
$$\text{on}(x, y) \wedge z \neq x \quad \rightarrow \quad \neg\text{on}(z, y) \qquad \text{— (2.3)}$$

(2.2) indicates that any object can be in only one place at any given time; (2.3) means that two different objects cannot be in the same position. Thus, if we add the fact:

$$\text{on(john, armchair)} \qquad \text{— (2.4)}$$

to the current world, the domain constraint (2.2) is violated and either (2.1) or (2.4) must be removed to form a possible world. Suppose, steve has already sat on armchair, the fact (2.4) must be rejected due to the constraint (2.3).

Truth maintenance [Do79] is a subject closely related to qualitative reasoning. Truth maintenance systems, such as [dK86], try to maintain some conditions based on known assumptions where each assumption has been given a confidence degree.

Although EDEN actions can be programmed to modify definitions using side-effects, the semantics of action procedures results in a poor representation of the knowledge. We may need to reconsider the syntactic structure of EDEN actions to improve the expressiveness.

2.6 Conclusion

The definitive paradigm has many characteristics in common with other programming paradigms. Some techniques of these programming languages could be usefully incorporated into definitive languages.