9
Conclusion

Definitive languages can specify objects abstractly using definitions. Uncertain parameters can be modelled as a set of definitions. For instance, F is the sum of two numbers A and B:

```
F Is A + B;
```

but A and B can be unknown or uncertain at the time of modelling. The definitions of A and B can be specified later when they are known. In addition, definitions can be added or modified afterwards.

The ability to model "uncertainties" distinguish definitive languages from conventional languages. The definitive languages are very suitable for computer-aided design purpose because "uncertainties" are frequently encountered by human designers. The interactive nature of definitive languages allows the user to modify definitions during a designing section.

In addition, the definitive programming paradigm has the following major advantages:

- **Expressiveness**: Relationships among objects are specified as mathematical expressions—*definitions*. The definitions can be specified in arbitrary order without affecting the final results.†
- **Machine independent**: The details of how the definitions are evaluated are hidden from the programmer. Depending on the implementation technique, the evaluations can be either eager or lazy, sequential or in parallel.
- **Efficient**: The mathematical expressions of the definitions implicitly specify the data dependency among variables. The definition manager

† This assumes that the functions have no hidden dependencies; otherwise, the evaluations of these functions may return unpredictable results. c.f. §5.2.

can make use of this information to reduce redundant evaluations. The definition manager may also take advantage of multiprocessor hardware because the data dependency information specifies how computations shall be synchronized.

- **Human/computer interaction**: Most, if not all, definitive systems allow the user to modify definitions incrementally. The user can query the current values of definitions by either giving a query command explicitly or inspecting the dialogue of values which are updated by the system automatically.

- **Software specification**: Since definitions are a very high-level mathematical abstraction of a program, they can serve directly as a specification. For instance, the interactive environment of DoNaLD gives the user the illusion of dealing with a specification of a picture rather than a program that draws it. The user modifies the definitions in a dialogue while watching the incremental change of the picture in another window.

The EDEN language combines two programming paradigms: the definitive programming paradigm and the imperative programming paradigm.

EDEN supports a very primitive form of definitions:

formula-variable-name **is** *formula-expression* ;

The functions/operators used in the *formula expression* are assumed to have no side-effect and hence no interference between the evaluation of definitions. In other words the only effect of evaluation of formula expression is to change the value of the definition. Therefore, the order of evaluation can be scheduled by the system automatically according to the implied data dependency.

Since the EDEN language is designed as an engine for implementing other definitive languages, it contains some imperative features to handle I/O interfaces which are procedural in nature. User-defined functions and procedures are also implemented using procedural statements in order to achieve good efficiency (since the EDEN interpreter is currently implemented on a von Neumann machine).

EDEN also introduces the concept of *action* which is a procedure invoked by the definition manager according to the data dependency information specified explicitly. An action specification has the form:

proc *action-name* : *variable-name* , ...
{ *procedural-statements* }

By associating actions (imperative procedures) with variables, the user can add monitoring routines to keep track of the relevant data without ever modifying the mathematical model (set of definitions and parameters). For example, the action print_a

```
proc print_a : a { writeln("a is now ", a); }
```

prints a message whenever a's value is changed.

Although an action can be regarded as an implementation of a definition, where the data dependency is explicitly specified and the formula expression is embedded in the action procedure body, it is, in principle, classified as another concept because it lacks the major characteristic of definition—*expressiveness*.

An action can serve very well as an independent output mapping *guarded action* which always stand ready for reflecting the up-to-date data to the user. By using actions to perform the I/O (or whatever), we can keep the mathematical model (the set of definitions) side-effect free. (Functional languages, on the other hand, have difficulties in handling I/O because the functional paradigm discourages the integration of side-effect and mathematical operation.)

In this thesis, we presented some definition management schemes of the definition manager. The current implementation of the EDEN interpreter is based on one of the schemes — *data dependency management* scheme. This scheme works fine in sequential systems. A concurrent evaluation scheme was presented but it needs detailed refinement.

EDEN is able to fulfil its aim — it is an engine of definitive notations. In this thesis, we showed how a subset of DoNaLD can be implemented using the EDEN interpreter as a back-end. Due to the lack of some handy features, like structured data types, macro definitions and higher-order definitions, it will be difficult to implement some sophisticated definitive notations or to do serious

programming. The future of EDEN is to upgrade the language to support these advanced features.

The opposite direction is to build a low level *definitive machine*, perhaps, in the form of a set of library routines which include all primitive definition management subroutines, so that definitive languages can be built directly on top of it. Thus, the intermediate EDEN language can be eliminated. Such a definitive machine may be based on the internal pseudo machine in the current implementation of the EDEN interpreter.

Because definitions implicitly specify data dependency which is important information in the analysis of parallelism detection, another promising research area is to support parallel computation in definitive languages.

In summary, the EDEN project has generated ideas related to many different areas — from programming concepts to implementation techniques. In this thesis, we have recorded as many ideas and experiences as possible and identified many issues requiring further research.