

Issues in Definitive to Procedural Translation

Nathan Griffiths
August 20, 1996



1

Motivation

There are several issues involved in the translation of a definitive program into a procedural one. However, before looking at these it is useful to consider why we might wish to do this. There are two main reasons why this kind of translation, or code generation, would be useful.

- Firstly, from the point of view of the Empirical Modelling Project, the conversion of a definitive script into a procedural program allows for circumscribing some of the definitions. This is equivalent to closing the open-world development of the script. A definitive script is very open-ended since any definition can be changed at run-time with a simple re-definition. This makes it an ideal prototyping tool. However there will come a point in the development at which some of the definitions will not need to be changed, and the functionality of the system is as desired. At this juncture it would be useful to be able to remove the facility to change the model. This might be done by introducing some kind of read/write permissions to definitive variables, however it could also be done by freezing definitions, making definitive variables into procedural ones, or in other words, performing a translation.
- Secondly, from a more practical and implementation point of view a procedural program should be more efficient than a definitive one, since there is no need for run-time definition maintenance to be done in a procedural program. Once a program is no longer in the development phase the need to be able to change the defining structure of the program is removed, thus the need for run-time definition maintenance is removed. It would be useful to take advantage of the performance

increase that this type of modification could provide.

There have been previous investigations into translation, all using roughly the same translation technique¹. Each attempt tries to analyse the dependency between variables and action calls. When a variable is reassigned, a sequence of reassignments and procedure calls follows it. In other words, the translator tries to predetermine the sequence of evaluations that **EDEN** will perform when a variable changes value, and then encodes this sequence into the translated program.

These translators suffer from two main limitations, namely that the evaluation sequence is completely predetermined and the user interface is not specified in the **EDEN** script. The problem with a predetermined evaluation sequence is that the translator cannot cope with changing dependency. Thus the interaction between user and resulting program is restricted to changing those variables with explicit values. In a sense this isn't a bad thing, since we are effectively closing the open-world development phase of the program. However, it is not unreasonable to assume that the user might actually need some definitive variables to be redefined after the translation—this is not possible if the evaluation sequence cannot be altered at runtime.

¹See “*Agent-oriented Modelling for Interactive Systems*”, Dr Y P Yung, 1996, Department of Computer Science, University of Warwick.

2

Translation schemes

There are a number of possible approaches to translation, each having different merits. Unfortunately there is not one singular method which possesses all of the merits, with none of the drawbacks.

2.1 The EDEN interpreter in C

This method simply wrappers the EDEN interpreter into an object library, which can then be called from a C program. The whole of the interpreter is linked into the program and since it is still present there is no increase in efficiency. This method doesn't really close the open-world development phase either, since any definitions can still be changed through calling the interpreter. It also requires a program to be re-written manually in C, rather than automatically translated. However this method does give the advantage of being able to write definitions in a conventional programming language.

2.2 Full analysis of consequences of actions

This approach is something of a non-starter, since the conditions of variables cannot be certain after a conditional statement, or on entry to a procedure (assuming the procedure can be called from more than one place in the program). To make the analysis possible some means of ensuring a variable is up to date is required, leading to the following scheme.

2.3 Generation of procedures for definition management

For each variable in the definitive script a set of small definition management routines are created. There are two types of variable in an EDEN script—implicitly defined and explicitly defined. If in a script we have ‘`a is b + c`’ then we say `a` is an implicitly defined definitive variable, conversely, if the definition was ‘`a = b + c`’ then `a` would be explicitly defined. In the former case if `b` or `c` were also implicitly defined then we say that `a` depends on a definitive variable.

Both implicitly and explicitly defined variables are translated through the introduction of definition management routines. The explicit case requires just two routines of *SET* and *MARK*. Implicit variables require an extra routine called *EVAL*. The *SET* procedure is used each time the variable is redefined (although redefinition of variables in procedural EDEN programs is not currently supported). *MARK* is used to set all dependent variable to be out of date, and *EVAL* is used to compute the value of the variable, i.e. make the variable up to date. The translation process has the effect of hard wiring the dependencies. These definition management routines are called whenever a variable used. By way of example consider the triggered function `result` in the guess program (See Appendices A and B for source code). The trigger variable for this function is `correct`. To translate the definitive description of `result` into procedural EDEN we need to introduce a flag to represent whether it is possible to evaluate the value of `correct` using procedural assignment. We call this flag `correctREADY`, and it is checked at the start of the procedural version of the `result` procedure. Assuming the value can be computed, we then use another procedure, `EVALcorrect`, which performs the procedural evaluation of `correct`. We can then continue with the `if (correct) . . .` statement. In (Figure 2.1) we can see the form of the translation from a definitive EDEN action to procedural.

The nature of a definitive script is such that definitions are not circumscribed, but are instead open-ended. When a definitive program is to be translated into a procedural program, the dependency structure must be tied down and the interface between the user and the set of definitions described. To do this a protocol is derived (from the LSD specification), and this protocol specifies the interactions that may take place between agents. To generate a procedural script from a definitive script we add a protocol, and then apply some standard utility functions. This protocol must

```

proc result : correct {
    if (correct)...
    ...
}

proc result {
    if (correctREADY == -1) return;
    EVALcorrect();
    if (correct)...
    ...
}

```

Figure 2.1: Definitive and procedural EDEN for `result`

be specified in the definitive script (by means of a comment) if automated translation is to be performed. In the procedural version it may be the case that there is a choice of actions which are applicable. If this is the case then the user will be prompted (using the predefined `user_input()` function) with a list of the actions from which the most desirable one should be chosen.

This approach is promising but still suffers limitations, for example the replay function in the billiards simulation does not work in the procedural translation.

The translator `def2proc` translates definitive EDEN into procedural EDEN by generating procedures for definition management. The expectation is that the resultant code should be more efficient, however this is not the case. The definitive version runs significantly faster than the procedural equivalent. The reason for this lies in the implementation of the EDEN interpreter. When executing a standard definitive script the definition management is handled by the definition maintainer, which is written in C and optimised (Figure 2.2). If the need for definition maintenance is removed by the introduction of EDEN procedures, i.e. the manner that `def2proc` functions, then there is considerably more EDEN code for the interpreter to handle. By the nature of the interpreter this has the net effect of a reduction, rather than an increase in speed.

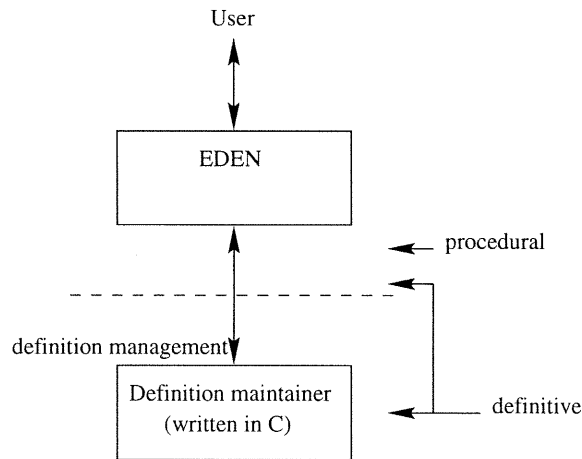


Figure 2.2: Definitive cf. procedural

2.4 Freezing variables

Variables can be frozen using the **EDEN** interpreter interactively. For example if there is a variable defined by `'a is b + c'` then its value will be frozen by passing `'a = a'` to the interpreter. This has the advantage of preventing redefinitions of frozen variable i.e. closing the open development environment, while still allowing unfrozen variables to be redefined in the normal manner. An increase in speed is dependent on the symbol table being updated to take account of a variables status as an explicit variable which no longer needs to be looked after by the dependency maintainer.

This approach isn't a true translation since there are many variables that have to remain implicitly (definitively) defined and the program is still interpreted and relies on the dependency maintainer. However if `tken` could provide an interface allowing a programmer to choose which variables could be explicitly defined, then this would be a useful tool. A list of variables could be presented by `tken` and the programmer could click on the ones that were to be frozen. It would not be realistic to implement a mechanism of automatically ascertaining which variable may be converted since there is no simple manner in which the semantics of a script can be analysed. The programmers knowledge is required to know which variables will not need to be redefined.

When a variable is frozen using the assignment method its entry in the

symbol table is updated. The variable changes status from being a `formula` to a `var` which doesn't require updating through the definition maintainer. This allows the program to run more efficiently. The graph in (Figure 2.3) shows that the version of the test program which had the frozen variables ran consistently faster than the pure definitive version.

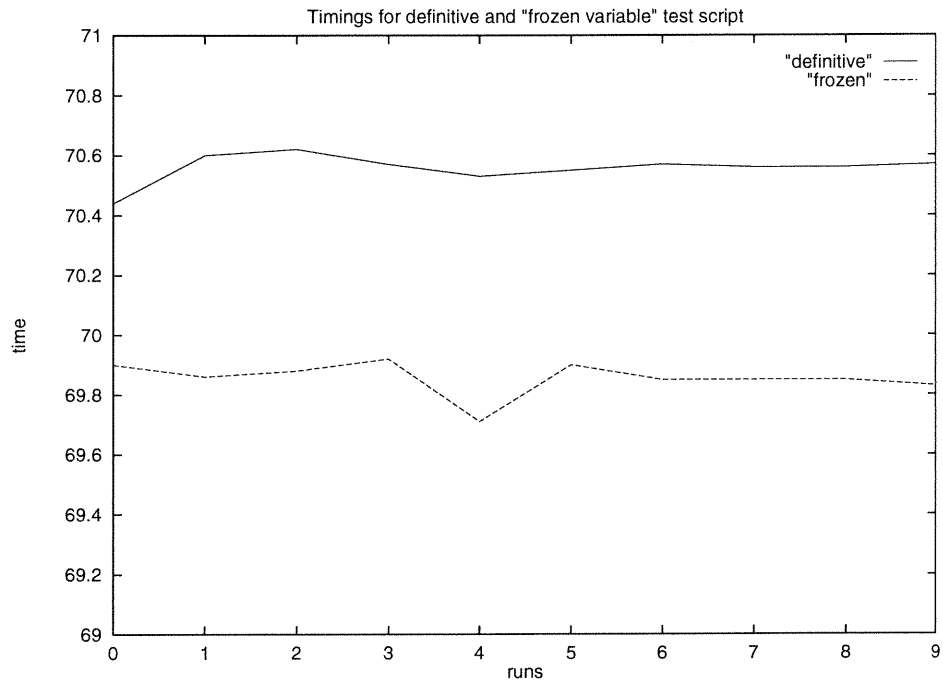


Figure 2.3: Program timings

3

Technical problems with C++

Let us assume that we will use the existing EDEN to procedural EDEN translator `def2proc`. This is a sensible assumption to make since the translator works for all of the simple test cases, and could be made to work for the more complicated cases such as the billiards simulation. A hand coded fix to the billiards simulation has been suggested, to allow the replay function to work as normal. The `def2proc` translator copes alleviates the need for a language shift in the early stages of translation, and deals wholly with the paradigm shift, namely from definitive to procedural. However the procedural language it translates into is not a typical procedural language and as such there are a number of problems that are encountered when the procedural EDEN is translated in a language such as C++.

3.1 General

There are several general problems with translating EDEN into C++. Some of these are relatively trivial to solve, while some are somewhat harder.

In EDEN variables don't have to be declared, while in C++ they do. This means that the translator would have to generate C++ code to declare any variables, and more importantly determine their type.

In EDEN a programmer can refer to a function name before it is declared, again this is not permitted in C++. For example, in the guess program (see Appendices A and B) `EVALcorrect()` calls `EVALsqrX()`, which is yet to be defined. This problem could be solved by declaring all functions which are referred to before their definitions at the top of the code. In fact there is no reason why all functions shouldn't be declared at the top of the code,

since this solves the problem, is easier to implement, and is no less efficient. To solve this problem a two pass compiler is needed, or at least a one pass compiler with a stack to push functions declarations onto.

In C++ a programmer must declare the arguments (parameters) of a function. However in EDEN this is dealt with at run time by the interpreter. This applies to the return type of a functions also. Eden sometimes uses the para aliasing, but even this contains no type information. For example in (Figure 3.1).

```
func sqr {  
    para x;  
    return x * x;  
}
```

Figure 3.1: The lack of type information in EDEN

This code snippet contains no type information, all we can deduce is that if x is an integer, or a float then the return type will be the same. EDEN doesn't support user defined types, which in this case is useful since if there were a user defined type of vector there would be no way to determine whether the return type was the same as the parameter type.

The protocol section in procedural EDEN is quite awkward to translate, since it requires a list in which the first and third elements are function, whilst the middle element is a string. This is discussed in the next section

3.2 Heterogeneous Lists

Heterogeneous lists can be implemented in C/C++ using `void*` pointers but we still need to know the type of a variable to be able to cast it to the correct type before use. Another problem is that if we had a linked list, the basic element would have to be large enough to hold to largest possible element. Consider the case where everything is an integer except one string of 1024 chars—this would be grossly inefficient if the list were large.

An alternative would be a linked list of structures, where each structure consisted of a data member (i.e. a `void*` pointer) and a type identifier. The type identifier could be a type from an enumeration declaring the type and the size of the data. The size is important to know how much memory should be allocated for a given piece of data.

```

/* a function with two distinct return types */
func testy {
    para x;
    if (x) return "x is true";
    return x;
}

/* function used to call testy() */
func test : x {
    writeln("-->",testy(x));
}

```

Figure 3.2: A function with multiple return types (string and integer)

```

box:q2[31] ttyeden ret_test.eden
1:> x = 0;
-->0
2:> x = 1;
-->x is true
3:>

```

Figure 3.3: Use of the function defined in (Figure 3.2)

3.3 Multiple Return Types

In EDEN a function declaration does not have to specify its return type, this information is determined by the interpreter at run-time. In C++, as in most other procedural languages (Pascal, C etc.), the return type of a function must be specified in the function declaration. Since the EDEN language is interpreted (at least in its current implementation) it is possible for a function to have a different return type in different situations. This is not possible in C++ since a function can only have a single, declared return type. An example of a simple EDEN function having multiple return types is given in (Figure 3.2) along with simple use in (Figure 3.3).

A possible fix to this is to place the returned value in a global variable, and have the function return the address (and type) of the variable. If we just had the return a `void*` we would not know which type the returned was, and so we would be unable to cast it for use. We could also have the functions return the type (eg through an enumeration) and data through variable that have be passed by reference. This would have the advantage that the data would not be in global scope, but only in the scope intended.

4

Scheme

An alternative approach is to aim for a translation from **EDEN** into a functional language such as scheme, or ML. ML still suffers from some of the problems that C and C++ present, the most important being that ML is a strongly typed language. This means that unless we create some user defined type the lists problem is still present, as is the function return type problem. Scheme on the other hand is not a strongly typed language. In the words of the MIT Scheme Reference Manual:

“Scheme has *latent* types as opposed to *manifest* types, which means that Scheme associates types with values (or objects) rather than with variables.”

This property means that such statements as `(define x (list 'a 3 'b))`, i.e. the list `(a 3 b)` are valid in scheme. While scheme solves the problems associated with heterogeneous lists, and other type related issues, it introduces as many problems as it remove. One of these problems is assignment. In a function language we cannot perform assignments to variables, as we can in **EDEN**. However, scheme is not a pure functional language, and does support assignment though the use of the `set!` operator. This has the disadvantage though of removing the mathematical cleanness of a program.

Scheme presents problems with respect to dependencies or triggered functions, since this is a property of a definitive language. One method for tackling this would be to use the scheme `set!` operator, and use scheme as though it were a procedural language. However, this would result in some very obfuscated code. The other approach would be to store the environment in a list (since there are no variables) and pass this list as an argument

to a function which calls itself. These functions would have the purpose of updating dependencies, and called triggered functions. This method would result in some very nasty code, which was grossly inefficient.

It seems that what at first sight seems like an appropriate choice of language is in fact worse than the original choice of C++. In C++ all of the difficulties are at the technical level of 'not quite' being able to do something. The difficulties in C++ could be overcome by using the language in a slightly non-standard manner. However the difficulties with scheme are at a higher level of the language simply not supporting the operations that we wish to be able to perform in our programs.

5

Conclusion

When both C/C++ and scheme are considered together it seems that C++ would be the best choice. C++ is not ideal by a long way for translating into from EDEN, however it is possibly more appropriate than scheme. It would, academically, be interesting to have a translation from EDEN to a functional language, possible scheme, but the resultant programs would most likely be next to useless due to their inefficiencies.

The big question is how do we convert EDEN into C++. It would be ideal if there were some program that we could pipe the output of `def2proc` through, which output C++. However this is non-trivial. We would need to make some changes to the interpreter so that it output the types of variables in a script, and perhaps also headers for the procedures and functions within the script. It may be better to (at least initially) opt for a compromise approach. I suggest that this comprise would take similar approach to `def2proc` where the user must specify extra information, perhaps again in the form of a comment. This has the advantage that the user must decide exactly on the functionality of the script, and explicitly close the development. If the programmer were made to give a type to each variable, and a header to each functions then the translation would become much easier. The main problems remaining are heterogeneous list and multiple return types of functions. The lists problem can be fixed through having a list class where each element is a structure containing a `void*` pointer to the data, and a type identifier. This is now feasible since we are being given the type information. The returning of a functions is perhaps best tackled through passing by reference of data and type information—since this enforces good scoping practise.

A

The Definitive Guess Program

```
func sqr {
    para x;
    return x * x;
}

sqrX is sqr(X);
correct is sqrX == target;

proc problem : target {
    writeln("X is the square-root of ", target, ". What is X?");
}

proc result : correct {
    if (correct)
        writeln("You've got it!");
    else
        writeln("The square of ", X, " is ", sqrX, ", not ", target);
}

/* initial setting */
target = 36;

/*protocol
correct -> target = sqr(rand() % 100);
!correct -> X = int(user_input("Enter X:"));
*/
```


B

The Procedural Guess Program

```
/***** Procedures for definition management *****/

proc SETcorrect { MARKcorrect(); correctREADY = 1; }
proc MARKcorrect {
  if (targetREADY == -1 || sqrxREADY == -1)
    return;
  correctREADY = 0;
}
proc EVALcorrect {
  if (!correctREADY) {
    if (!sqrxREADY)
      EVALsqrx();
    correct = sqrx == target;
    correctREADY = 1;
  }
}

proc SETtarget { MARKtarget(); targetREADY = 1; }
proc MARKtarget {
  targetREADY = 0;
  MARKcorrect();
}

proc SETsqrx { MARKsqrx(); sqrxREADY = 1; }
proc MARKsqrx {
  if (XREADY == -1 || sqrxREADY == -1)
    return;
  sqrxREADY = 0;
  MARKcorrect();
}
proc EVALsqrx {
```

```

    if (!sqrXREADY) {
        sqrX = sqr(X);
        sqrXREADY = 1;
    }
}

proc SETX { MARKX(); XREADY = 1; }
proc MARKX {
    XREADY = 0;
    MARKSqrX();
}

proc SETSqr { MARKSqr(); sqrREADY = 1; }
proc MARKSqr {
    sqrREADY = 0;
    MARKSqrX();
}

/***** translations of EDEN procedures and functions *****/

func sqr { para x;
    return x * x;
}

/***** translations of EDEN actions *****/

proc result {
    if (correctREADY == -1) return;
    EVALcorrect();
    if (correct)
    {
        writeln("You've got it!");
    }
    else
    {
        EVALSqrX();
        writeln("The square of ", X, " is ", sqrX, ", not ", target);
    }
}

proc problem {
    if (targetREADY == -1) return;
    writeln("X is the square-root of ", target, ". What is X?");
}

```

```

/***** Initial Status EDEN variables:
-1 = undefined, 0 = outdated, 1 = up-to-date *****/

correctREADY = -1;
target = 36;
targetREADY = 1;
sqrXREADY = -1;
XREADY = -1;
sqrREADY = 1;

/***** Consequence of the initial definitions *****/
{
problem();
result();
}

/***** Translation of the external triggering variables *****/

/***** Translation of the protocol *****/

func guard1
{
    EVALcorrect();
    return correct;
}

proc action1
{
    target = sqr(rand() % 100);
    SETtarget();
    problem();
    result();
}

func guard2
{
    EVALcorrect();
    return !correct;
}

proc action2
{
    X = int(user_input("Enter X:"));
    SETX();
    result();
}

```

```

}

protocol = [
  [
    guard1,
    "target = sqr(rand() % 100);
",
    action1
  ], [
    guard2,
    "X = int(user_input(\"Enter X:\"));
",
    action2
  ]
];

proc USER_INPUT {
  auto i, choice, actionList;
  for (;;) {
actionList = [];
for (i = 1; i <= protocol#; i++) {
  if (protocol[i][1]())
/* append actionList, i; */
actionList = actionList // [i];
}
if (actionList# > 1) {
  /* have choice */
  writeln("Possible actions:");
  for (i = 1; i <= actionList#; i++) {
writeln(i, ": ", protocol[actionList[i]][2]);
  }
  writeln("Enter your choice:");
  choice = int(gets());
  if (choice < 1 || choice > actionList#)
continue;
  else
protocol[actionList[choice]][3]();
} else if (actionList# == 1) {
  protocol[actionList[1]][3]();
}
}
}

func user_input { para message;
write(message);

```

```
return gets();  
}
```

```
USER_INPUT();
```

